ҚАЗАҚСТАН РЕСПУБЛИКАСЫ
БІЛІМ ЖӘНЕ ҒЫЛЫМ МИНИСТРЛІГІ

Ахметова Майра

ФУНКЦИОНАЛДЫҚ-ЛОГИКАЛЫҚ ПРОГРАММАЛАУ ЖӘНЕ ЖАСАНДЫ ЗЕРДЕ ЖҮЙЕЛЕРІ

Алматы, 2012

УДК 004.4(075.8)
А 94

Бастау баспасының 2012 жылғы жоспары бойынша басылды АЭжБУ Ғылыми кеңесі «Есептеу техникасы бағдарламалық қамтамасыз ету» мамандығы бағыты бойынша оқу құралы есебінде ұсынған

Пікір берушілер:
Д.Н.Шоқаев – ҚазҰТУ, техн. ғыл. д-ры., проф.
Л.Қ.Ибраева – АБжЭУ, техн. ғыл. канд., проф.

А 94 Ахметова М. Функционалдық-логикалық программалау және жасанды зерде жүйелері: Оқу құралы / М.Ахметова. – Алматы: «Бастау» баспасы. – 2012. – 330 бет.

ISBN 978-601-7275-38-9

Оқу құралында жасанды зерде жүйелері мен оның программалау тілдерінің теориялық мәселелері қарастырылған. Программалаудың мағлұмдамалық стилінің өкілдері – Лисп, Пролог, Хаскел тілдерінің мүмкіндіктері баяндалған.

Оқу құралы 5В070400 – «Есептеу техникасы бағдарламалық қамтамасыз ету» мамандығының бағыты бойынша оқитын студенттерге арналған.

Без. 20, Кесте 6, әдеб. көрсеткіші – 50 атау.

УДК 004.4(075.8)

ISBN 978-601-7275-38-9

© Ахметова М., 2012.
© Бастау» баспасы, 2012.

АЛҒЫ СӨЗ

Қазіргі ғаламтор мен ақпараттық технологиялар заманында дүние жүзіндегі елдердің, соның ішінде біздің еліміздің де есептеу техникасы және программалаушы мамандарға деген сұранысы артып отыр. Осы мамандықтарды меңгеретін оқушылар үшін жасанды зерде (искусственный интеллект) пәнінің алатын үлесі де аз емес. Бұл кітап өз оқушыларына қасиетті ғылым саласының осы жасанды зерде бағытында қазақ тілінде ұсынылып отырған тұңғыш оқу құралдарының бірі десек, қателеспейміз.

Оқу құралында жасанды зерде саласының негізгі іргелі бөліктерімен қатар, оның құралдарына де көңіл бөлінген. Жасанды зерде жүйелерін жасауда оның құралдарының атқаратын рөлі зор. Компьютермен тілдесетін программалау тілдерінде білімдер көбінде нұсқаулар тізімі-командалар сияқты сөйлем тізбектерінен тұрады. Қазіргі кеңінен қолданылып жүрген программалау стильдері осы тәсілді қолданады. Ал жасанды зерде құралдарының тілдерінің негізі болатын логикалық және функционалдық стильдер мәселе сипатын баяндау тәсілімен береді. Кітапта осы екі бағыттың математикалық теориясымен қоса, осы стильдің көрнекі өкілдері – Пролог, Лисп, Хаскелл тілдерінің негіздері келтірілген.

Адам миының ең ғажап қасиеттерінің қатарына ол бір нәрсені үйреніп алған соң, басқа жағдайларда үйренбеген әрекеттерді орындауы жатады. Жасанды нейрон желілердің де осы үйренуге деген қабілеті бар болып шықты. Олардың бұл қасиеті адамның үйрену қасиетіне ұқсайтыны соншалық, тіпті адам баласы осы үдерісті тереңінен түсініп, оны қайталау мүмкіндігіне ие болды ма деген ойлар келуі де мүмкін. Бірақ ол әзірге тоқмейілсу болып қалуда. Өйткені жасанды нейрон желілерінің үйрену қасиеттерінің де белгілі бір шегі бар болып шықты. Ол адамның үйрену деңгейіне шығуы үшін әлі де көптеген күрделі сұрақтарға жауап табуы қажет. Дегенмен қазірдің өзінде нейрон жүйелерінің нәтижелерін қолдану көптеген сала аумақтарында экономикалық, техникалық жетістіктерге әкелуде. Оқу құралында осы нейрон жүйелеріне негізделген үйрету жүйелерінің зерттеулері келтірілген.

Сонымен бірге кітапта жасанды зерде жүйелерін құру бағытындағы жаттығулар мен бақылау сұрақтары да бар. Осы салаға қатысты терминдердің түсіндірмелері да қамтылған. Оқу құралындағы ма-

териалдар автордың көп жылдар бойы АЭжБУ, ҚазҰТУ және басқа да жоғарғы оқу орындарында оқылған дәрістер мен зертханалық жұмыстары негізінде құрастырылған.

Кітаптағы ақпараттану саласына қатысты терминдердің көпшілігін біз терминком бекіткен сөздіктен алдық.[1]

Қазақша-орысша, орысша-қазақша терминологиялық сөздік. 3 том, Информатика және есептеуіш техникасы. Алматы: "Рауан", 1999.

КІРІСПЕ

Қазіргі таңда программалық қамту құрамы күн өткен сайын күрделене түсуде. Бүгінгі таңдағы программалау тілдері индустриясының назары белгілі бір қасиеттерге ауысты. Қазіргі кезде программаны орындау жылдамдығы немесе супер оңтайлы коданы жазу мүмкіндігі ешкімді таңғалдыра алмайды. Бүгінгі таңда алға қойылып отырған талаптардың бірі – программа құрушының өзі құрған кодасынан өзі шыға алмайтын қиындықты жеңу немесе бір не бірнеше жыл өткен соң, өзінің не жазып не қойғанын білмей бас қатыруы емес, қайта осы көрсетілген кедергілерді жоя білу қабілеті жатады. Сондықтан программалау тілдеріне деген талаптардың ең маңыздысы және күрделісіне бұндай тілдердің табиғи тілге жақындығы жатады.

Ғылыми зерттеулерді автоматтандыру ісінің тарихына көз жіберсек, онда негізгі екі кезеңді атап өтуге болады: *біріншісі* – мәліметтерді жинау барысын автоматтандыру, көп есептеуді қажет ететін, бірақ шығару жолы оңай түрін ЕЭМ салу, сызықтық программалау әдістерін қолданып, тиімді шешім табу есептерін машинаға орындату; *екіншісі* – зерттеудің қисынды шешімдерін автоматты талдау мен жіктеуден өткізу. Осы уақытқа дейінгі жүргізілген жұмыстар мен зерттеулердің көпшілігі бірінші кезеңнің есептерін шешіп келді. Қазіргі уақытта жасанды зерденің әдістерін пайдалану мен екінші кезеңнің есептерін шығару мүмкіндігін беріп отыр. Басқару теориясындағы есептерді шешуге арналған амалдар санының күрт көбейіп кетуі және есепті құрудың өзі оны шешу тәсілдері қажет ететіндей дәрежеде болмауы әдеттегі қолданылатын әдістердің тиімсіздігін көрсетті. Жалпы жағдайда программалау тілдері алгоритмдік программалау, құрылымдық программалау, объектті бағытталған программалау сияқты көптеген кезеңдер мен стильдерден өтті.

Міне, осы жағдайда жасанды зерде бағытындағы зерттелген тәсілдер мен ой-пікірлер көмекке келеді. Мұндай көмекті аса қажет ететін басқару жүйелері қатарына: құрамындағы есептерді шешуде тек адамдарға ғана тән, солар ғана орындай алатын *интеллектуалдық*[1] деп аталатын амалдары бар жүйелерді жатқызуға болады. Әдеттегі тәсілдерді қолданып шығаратын есептер шешу жолдары көрсетілген

[1] Интеллектуалдық есептерді, әдетте, адам баласына қатысты орындалатын шығармашылыққа жатқызады. Ал компьютер атқаратын мұндай есептерді жасанды зерде саласы техникалық және программалық жүйелер көмегімен орындайды.

қатаң тәртіппен тізбектелген алгоритмдерге сүйенсе, эвристикалық есептердің деректері сарапшы адамдардан алатын мағлұматқа сүйенеді. Ол сан түрінде ғана емес, сапалы түрде де қарама-қайшылыққа толы болуы мүмкін. Осы жағдайда жасанды зерде тәсілдерін қолдануға болады. Олар сандармен ғана емес, құрылым сапасы әртүрлі білімдермен жұмыс істейді. Бұл тәсілдер мамандар білімін жинақтауға, топтауға, жіктеуге, оларды басқару әдістерімен байланыстыруға мүмкіндік береді. Осы тәсілдер көмегімен құрылған программалық құралдарды ЭЕМ мен пайдаланушы адам арасындағы зияткерлік интерфейс есебінде қарауға болады. Бұл байланыс қазірдің өзінде белгілі бір саладағы мамандардың білімін пайдалануға мүмкіндік береді. Мұндай жинақталған білім мен тәжірибе алмасу есептері құрылым жағынан өте қиын болып келетін саладағы мамандар үшін өте құнды. Жасанды зерденің тәсілдерін қолданып, осындай білім мен тәжірбие жинақтаған программалар *сарапшы жүйелер* деп аталады. Бұл жүйелердің құрамында білімнің жинақтаған қоры мен сонымен жұмыс істейтін арнайы басқару программалары бар. Бұл жүйелерді кез келген пайдаланушы адам өз мәселесін шешуге қатыстыра алады. Ол үшін жүйеге түсінікті болатын, мамандар қолданатын терминдері бар сұрақтар беру арқылы жауап ала алады. Бұл жүйедегі тәсілдер жасанды зерде теориясының бір бағыты есебінде ары қарай дамып отыр.

Жасанды зерде тілдерінде бағдарламалау әдістері функционалдық және логикалық программалау тілдерімен тығыз байланысты. Бұндай тәсіл программалаудың мағлұматтық стиліне жатады. Лисп, Haskell және Пролог – тілдері осындай программалау стилінің көрнекі өкілдері. Бұл тілдерге қойылатын талаптарға мыналарды жатқызамыз: тілдер құрылымы компьютерлік архитектураның төменгі деңгейімен, операциялық жүйелер және аппараттық құралдар архитектурасы[1] деңгейімен, зерде көлемі, процессор жеделдігімен шектеулі болуы қажет. Жасанды зерде жүйелері әдетте басқа ірі жүйелердің модульдері ретінде құрылады. Сондықтан осы ірі жүйелерге қатысты барлық стандарттық талаптар тілге де қатысты болады. Соған қарамастан, функционалдық және логикалық программалаудағы модульдер көптеген прототиптер мен жаңа әдістерді меңгеруге қажетті құралдарды пайдалана алады. Сонымен бірге бұл тілдер жаңа заманғы

[1] Бұл жерде «архитектура» программалық қамтамаға (ағылш. software architecture) қатысты айтылып отыр. Ол құрамында сыртқы түрі көрінетін программаның құрамдас бөліктері мен олардың арақатынасын сипаттайтын программаның не есептеу жүйесінің құрылымы.

программалау тілдеріне енетін жаңа құралдар негізін құруда көп рөл атқарады. Қазіргі заманғы программалаудағы дүниежүзілік тәжірибе ЖЗ тілдеріндегі стандарттан үйренуден жалыққан жоқ. Осындай мақсатты іске асыруда болашақта функционалдық және логикалық программалау тілдерінің маңызы арта түседі деп есептейміз.

«Функционалды логикалық программалау және жасанды зерде жүйелері» оқу құралының мақсатына жасанды зерде программалау тілдерінің теориялық негіздері, жұмыс істеу принциптері және әдістерін оқып білу жатады. Білімнің мұндай түрін оқып үйрену жасанды зерде саласындағы есептерді шешуге мүмкіндік береді.

Программалаудың бұл стилінде жағдайды сипаттауға көп көңіл бөлінеді. Программалау мақсат термині төңірегінде өтеді. Сала мәселесін қою жағдайды анықтаудан басталып, одан ары қарай есепті қалай шешу туралы бүге-шүгесіне дейін сипатталады. Яғни әдеттегі программалау тәсілдеріндегідей не істеу керек екені көрсетілмейді.

Сонымен қатар оқу құралында жасанды зерденің әдістері мен үлгілірін қолдана білу дағдысын да үйрету, студенттерді іс жүзіндегі ЖЗ жүйелерін құруға арналған программалық қамту жасау технологияларына оқыту әдістері келтірілген.

Жасанды зерденің әдістері есепті шығарудағы негізгі іздеу жолдарын зерттейді, яғни зердені пайдаланып, шешімді табу стратегиясын оқытады. Оқу құралында жасанды зерде есептерінің негізгі жүйелеу принциптері, кеңістіктегі күйі бойынша және есептерді бөлшектеу күйі бойынша іздейтін әдістер қарастырылады. ЖЗ теориясы мен практикасы дамуының нәтижесі болып табылатын сарапшы жүйелерді жобалаудың ерекшеліктері де қарастырылады.

Оқу құралының алдына қойған мақсаттарының біріне қазіргі жасанды зерде саласындағы есептерді шығаруда программалаудың мағлұмдамалық стилін қолдану, алынған нәтижелерден дұрыс түсіндірмелерін беруді үйретумен бірге функционалдық және логикалық программалаудың, предикаттарды есептеу теориясының негізгі ұғымдарын, Лисп, Хаскелл және Пролог тілдерінің математикалық негіздерін, осы тілдердің синтаксисі мен семантикасын оқыту жатады.

Кітапты оқу нәтижесінде оқушы ЖЗ негізгі тұжырымдамасымен танысады. ЖЗ аумағындағы негізгі мәселелер мен оны шешу жолдарын оқып үйренеді. Білімдерді пайдаланып, жобалаудың негізгі үлгілерін оқумен бірге зерделік жүйелерді құрудың негізгі көзқарастары мен оның математикалық, программалық құралдарымен танысады және

сарапшы адамның білімдерін ала білу жолдарымен танысады, маман білімін қалыпқа келтіру тәсілдерін меңгереді, сарапшы және зерделік жүйелердің әдістері мен үлгілерін қолдана білу дағдысын үйренеді, осы жүйелердің программалық қамту[1] технологияларын меңгереді. Сонымен бірге жасанды зерде тілдерінің негізін қалайтын функционалдық және логикалық программалау тілдерінде программалық нысандарын танып біледі, деректер құрылымдарымен жұмыс істеуге, оларды өзгертуге, Лисп, Хаскелл және Пролог тілдеріндегі рекурсия механизмін қолдана білуге, осындай құрылымдарды қазіргі жасанды зерде саласындағы есептерді шығаруда қолдана білуді үйренеді.

[1] Программалық қамтама (обеспечение) – жасалған программалық өнімдерді іске қосуға қажетті программалық жүйелер жиынтығы.

1-бөлім. ЛОГИКАЛЫҚ ПРОГРАММАЛАУ

Бұл тақырыптың негізгі мақсаты – жасанды зерде (ЖЗ) есептеріне арналған программалау тілдерін таңдауға байланысты негізгі мәселелерді қарастыру. ЖЗ есептерінің негізгі мақсаты зерделік есептерді шешуге қажетті басқару және деректерді көрсету құрылымдарын қалыптастыру болып табылады. Осындай мақсатты іске асыру тілдеріне қажетті қасиеттерді, негізінен, анықтайтын – осы құрылымдарға қойылатын талаптар. Зерделік жүйелерді сипаттау екі түрлі деңгейде өтеді: білімдер және таңбалар. Жасанды зерде тілдерінде программалау әдістері функционалдық және логикалық программалау тілдерімен тығыз байланысты. Бұндай тәсіл программалаудың мағлұмдамалық стиліне жатады. Лисп, Хаскелл және Пролог – тілдері осындай программалау стилінің көрнекі өкілдері. Бұл тілдерге қойылатын талаптарға мыналарды жатқызамыз: тілдер құрылымы компьютерлік архитектураның төменгі деңгейімен, операциялық жүйелер және аппараттық құралдар архитектурасы деңгейімен, зерде көлемі, процессор жеделдігімен шектеулі болуы қажет. Жасанды зерде жүйелері әдетте басқа ірі жүйелердің модульдері ретінде құрылады. Сондықтан осы ірі жүйелерге қатысты барлық стандарттық талаптар тілге де қатысты. Қазіргі кезде ЖЗ жүйелері Smalltalk, C, C++ және Java сияқты тілдерде іске асырылған. Осындай пайдаланудың мысалы ретінде Java тілін атауға болады. Бұл тілде ЖЗ есептерінің программалау тілдерінде ең алғаш қолданылған серпінді байластыру, зердені автоматтық басқару сияқты көптеген жаңа құралдар пайдаланылған. Соған қарамастан, Лисп, Хаскелл және Пролог тіліндегі программалар көптеген прототиптер мен жаңа әдістерді меңгеруге қажетті құралдар болып отыр. Сонымен бірге бұл тілдер жаңа заманғы программалау тілдеріне енетін жаңа құралдар негізін құруда көп рөл атқарады. Қазіргі заманғы программалаудағы дүниежүзілік тәжірибе ЖЗ тілдеріндегі стандарттан үйренуден жалыққан жоқ. Осындай мақсатты іске асыруда болашақта Лисп және Пролог тілдерінің маңызы арта түседі деп есептейміз.

Ең бірінші Пролог тіліндегі программа 1970 жылдың басында Францияда құрылды. Ол табиғи тілді түсіну жобасы көлемінде жасалды. Пролог тілінің негізгі даму кезеңі 1975-1979 жылдарға келеді. Бұл кезеңде тілдің іске асуына Эдинбург университетінің жасанды зерде кафедрасының қызметкерлері Дэвид Уоррен (David H.D. Warren) и Фернандо Перейра (Fernando Pereira) жауап берді. Олар Пролог тілінің

ең бірінші түсіндіргішін құрды. Бұл программалық өнім DEC 10 жүйесі негізінде жасалды. Сонымен бірге ол түсіндіргіш режимінде де, компилятор[1] режимінде де жұмыс істей алатын мүмкіндігі бар екенін көрсетті. Прологтың осы нұсқасы оның ең бірінші стандарты болып табылады.

Логикалық программалаудың ерекшелігін арттыру оның құндылығын бағалайтын зерттеу жобаларын құрғанда пролог тілінің артықшылығын кеңінен көрсетеді. Осы тілде жасалған көптеген қолданбалар ЖЗ мәселесіне арналған Халықаралық конференция мен Логикалық бағдарламалауға арналған симпозиум еңбектерінде жарияланған [3].

Пролог программалаудың қисынды жүйедегі тіліне жатады. Пролог тілін сарапшы жүйе – бағдарламалау бағытындағы кеңінен қолданылатын тілдер қатарына жатқызуға болады. Бұл – басқалармен салыстырғанда жас тіл, оның негізінде математикалық қисын әдістері жатыр. Бұл дербес компьютерлерге арналған сарапшы жүйелер, білім қоры, табиғи тілмен жұмыс істейтін жүйелер сияқты программалар құруға мүмкіндік береді. Пролог тілі жоғарғы деңгей тіліне жатады. Оның қатаң түрде жүйеленген теориялық негіздемесі бар. Ол математикалық логиканың тұжырымдамасы мен әдістерін пайдалануға бағытталған. Яғни Пролог логика терминдерінде программалауға арналған. Прологтың басқа тілдерден ерекшелігі – онда жазылған программалардың мағлұмдамалық сипатта болуы. Программаның негізгі құрылымдық блоктарына белгілі бір құрылымдағы нысандар жиыны және осы нысандарды байланыстыратын функциялар мен қатынастар жатады. Прологта әдеттегі пограммаларда кездесетін шартты операторлар, кезең немесе өту операторлары сияқты басқару конструкциялары болмайды. Пролог программасы шығаратын мәселенің үлгісі есебінде болады. Пролог программалау, ойлау стилінің басқа түрін қажет етеді. Сондықтан кейде мұндай жұмыс психологиялық белгілі бір күш салуды қажет етеді. Яғни әдеттегідей мәселені шешу үшін белгілі бір іс-әрекет тізбегін берудің орнына, Пролог программасында нысандар мен олардың қарым-қатынасы терминдері арқылы есептің мазмұнын беру қажет. Программалаушы Пролог та есеп алгоритмінің орнына, оның логикалық спецификациясын құрады. Ал

Компилятор – программалау тілідерінің бастапқы кодаларын машина тіліне аударатын құрал. Бұл кодаларды іске қосу үшін, тағы белгілі бір программалар қажет. Бұл – компилятордың интерпретатордан айырмашылығы.

ары қарай алгоритм құруды Пролог жүйесі автоматты түрде өзі жасайды. Ол оны өз бойындағы логикалық шығару механизмі көмегімен іске асырады. Бұл іс-әрекет былайша атқарылады: мәселенің мақсаты дерекқорға сұрату есебінде құрылады. Бұл сұратуда салалық мәселе аумағының сипаттамасы болады. Сұратудағы мәндерді дерекқордан іздеу үшін Пролог жүйесі шығару механизмін іске қосады. Осылайша Прологта өтетін есептеулер есептің мақсатты тұжырымын дәлелдеуге арналған дедукция үдерісі болып табылады. Қазіргі кезде Пролог тілі де, оның негізіндегі теориялық тұжырымдамалар да көптеген салаларда кеңінен тарап, етек жайды. Ондай сала қатарына: *есептеу жүйелері архитектурасы*, программалау тілдері семантикасының *формальды сипаты*, дедуктивтік мүмкіндігі бар дерекқорлар сарапшы жүйелерді құру, лингвистика сияқты сала мәселелері жатады.

1.1. Логикалық программалау тарихы

Логикалық программалау идеялары 1930 жылдардың басында пайда бола бастады. Сол жылдары француз математигі Жак Эрбран (1908- 1931) ұсынған теорема логикалық бағдарламалау зерттеулерінің негізі десек те болады. Бұл зерттеуді теореманы автоматты түрде дәлелдеудің (АДТ) алгоритмі деп атайды. Ол бұл зерттеу нәтижесін «1-ші дәрежелі предикаттарды есептеуге» қолданды. Осыдан 30 жыл өткенде, ең алғаш рет Эрбран теоремасын американдық ғалым Ван Хао іске асырды.

1965 жылы Дж.Робинсон Эрбран алгоритмін ЭЕМ лайықтап өңдеп, толықтырып, оның унификацияланған тиімді түрін құрды. Оны ол резолюция принципі деп атады.

Логикалық программалау идеяларын іске асыруға Массачутес технологиялық институтының ғалымдары Хьют пен Зусман өз үлестерін қосты. Олар 1960 жылдары Плэнер (Planner) және Микро-Плэнер (Micro-Planner) тілдерін құрды.

1960-1970 жылдары логикалық программалау теориясы Роберт Ковальский зерттеулерімен толықты. Ол 1974 жылы «Предикаттар логикасы – программалау тілі» деген еңбегінде ол қисынды есептеуде логикалық өрнектер орнына хорн дизъюнктері деп аталатын формуланы пайдалануды ұсынды.

Логикалық программалау зерттеулері одан ары қарай да іргелі ізденістерден құралақан болған жоқ. 1973 жылы Марсель университетінің Ален Колмэро бастаған бір топ ғалымдары резолюция прин-

ципіне табан тірей отырып, теореманы дәлелдейтін программа құрды. Бұл программа негізінде Ковальский жасаған түсіндіргіш тұрды. Ол табиғи тіл мәтіндерін жөндеп, өңдеу жүйесіне қызмет етті. Программа Фортран тілінде жазылды және ол өте баяу жұмыс істеді. Кейіннен бұл өнімді Пролог (Prolog – PROgrammation en LOGique) деп атады.

1976 жылы Р.Ковальский мен Маартен ван Эмден (Эдинбург университеті) логикалық программаларды құрудың екі түрлі жолын ұсынды: процедуралық және мағлұмдамалық.

Іргелі теориялық жұмыстар мен көптеген қызықты ой желілері туып жатса да, логикалық программалау тұжырымдамасы әсіресе технократ АҚШ-қа қатысты кеңінен даму таппады. Ол тек академиялық ортада қолданыс тауып отырды. Мысалы, 1977 жылы Д.Уоррен және Ф.Перейра (Эдинбург университеті) ***DEC PDP-10*** ЕЭМ арналған Пролог тілінің тиімді интерпретаторын[1] жасады. Интерпретатор толығынан Пролог тілінде құрылды. Тілдің осы іске асырылу жолы «Эдинбург нұсқасы» деген атаумен көптеген жылдар бойы тілдің басқа нұсқаларына түптұлға етінде қызмет етіп, тілдің ресми емес стандарты қызметін атқарды.

Болашақты қамтитын маңызды идеялары мен сәтті орындалған жолдары бола тұрса да, жасанды зерде бағытында зерттеулер жүргізіп отырған батыстың көптеген ұйымдарының логикалық бағдарламалауға көзқарастары немқұрайды болды. 1980 жылдары Прологты зерттеушілер саны бүкіл дүние жүзі бойынша жүзге де жетпейтін еді.

Жағдай 1981 жылдың қазан айынан бастап күрт өзгерді. Өйткені Жапон елі бесінші буынды есептеу машинасын құратыны туралы дүние жүзіне мәлімдеме жасады. Онда Пролог тілі осындай машиналардың негізгі базалық тілі есебінде қабылданған болатын. Осы 1981 жыл бойы Пролог тақырыбы көптеген ғылыми журналдар мен жасанды зерде бағытындағы зерттеулер жарияланулары беттерінен түспеді. Пролог тілі өзінің белгілі бір биіктігіне көтеріліп, оның стандарты енгізіле бастады. 1990 жылы Прологтың ***ISO/IEC 13211 – 1:1995*** деген стандарты пайда болды.

Аталған «жапон жобасының» мақсаты аса айқын болмаса да, ондағы логикалык бағдарламалаудың маңызды рөлі анық бейнеленіп көрсетілді. Бұл жобаның негізгі мақсаты ***білімдерді өңдейтін жүйе*** құру болатын. Жүйені құрушылар жүйе пайдаланушымен қарымқатынасты табиғи тілде жүргізу арқылы басқарса, ал программала-

---

[1] Интерпретатор дегеніміз – ол программа операторларын немесе командаларын талдап, оны бірден орындайтын программа немесе аппараттық құрал.

ушы маманмен логикалық программалау тілінде іске асырады деген мақсатты көздеді. Бұндай жүйелер өздерін өздері үйретумен бірге, өздеріндегі бар білімді жаңалармен толықтырып отыру қасиетін де меңгеруі қажет еді. Жапондар ЭЕМ жұмысын адамның ойлау деңгейіне көтеру мәселесін қойды. Олар фон-Нейман архитектурасынан бөлек деңгейдегі архитектура құруды көздеді. Жапон жобасынан соң американдық және еуропалық жобалар да дүниеге келе бастады. Осындай мақсаттарға лайықталған ***PSI*** және ***PIM*** деген логикалық программалау компьютерлері құрылды.

Осыдан төрт жыл өткен соң, ***Delta*** деген атты ЭЕМ құрылды. Онда логикалық шығару машинасы, реляциялық дерекқор және табиғи тілдегі интрефейс болды. Бұл машина қуаты 20 Гбайт болды. Мұндай машинамен жұмыс істеу кезінде дерекқордағы операцияларды қатар атқару қажеттігі мен жұмыс атқару жылдамдығының миллиард липске жетуі анықталды. Мұндағы липс дегеніміз – бір секунд аралығындағы логикалық қорытынды шығару қадамдары. Бұл мақсатты орындау үшін қатар жұмыс атқарып тұратын 100 процессор қажет болды. Бұндай 100-ядролық процессорлар жапон елінде 1990 жылдар басында болады деген сенім бар еді.

Қазіргі таңда көп ядролы процессорлардың кеңінен шығарыла бастауы логикалық программалау мен Пролог тілінің қатарлас есептеулермен біріге отыры, аса қажетті ғылыми зерттеулердің жаңа деңгейлеріне көтерілуі осыдан болса керек.

Сонымен, бесінші буынды ЭЕМ жасау жобасы жасанды зерде саласындағы алғашқы жобалардай тағы да тоқырауға ұшырап жабылды. Бірақ оған кеткен қаншама еңбектер мен зерттеулердің барлығы босқа болды деген ойдан аулақпыз. Жобалар нәтижесінде білімді пайдалану және логикалық қатарлас қорытынды шығару қисындарының зерттеулері мен жинақталған тәжірибелер жасанды зерде саласында жаңа ойларға қозғау салып, логикалық программалау әдістеріне қызығушылық тудырды.

Қазіргі таңда Пролог тілі жапон елі мен Еуропа елдерінде жасанды зерденің ең кең тараған тіліне саналады. Ал АҚШ-та жасанды зерденің көп тараған тіліне функционалдық программалаудың тілдері жатады.

Қазіргі кезде Пролог тілінің іске асырылған 30-ға жуық түрлері бар. Соның ішінде «коммерциялық» деген атпен белгілі Пролог жүйелерге: ***Quintes Prolog*** және ***SICStus Prolog*** жүйелер жатады. Бұл жүйелерде көптеген құралдар мен кең көлемді кітапханалар бар және оны ***SICS (Swedish Institutebof Computer Science)*** қолдайды. Жылдамдығы жағынан осы ***SICStus*** жүйесінен кем түспейтін және екілік кода-

лы компиляторы бар ***GNU Prolog*** шығарылды. Кейбір кеңейтулері бар, байт-кодқа негізделген, іске асырылған жүйелердің ең жылдамы ***B-Prolog*** жүйесі бар. Пакет түрінде құрылған ***Strawberry Prolog*** жүйесі Пролог тілімен танысуға және ***Windows*** және ***Linux*** ортасында шағын бағдарламалар жазуға жарайды. Дегенмен бұл жүйе күрделі бағдарламаларды жазуға жарамайды.

Тағы бір атап өтуге жарайтын шағын компьютерлерге арналған ***Palm Toy Language Palm*** деп аталатын Пролог түрін айтуға болады. Қазіргі кезде кеңінен тараған Пролог жүйелерінің біріне ***SWI-Prolog*** жүйесін жатқызуға болады. Онда ***XPCE*** графикалық құралдары және оның ***GNU*** лицензиясы бар. Бұл шығарылатын ***GNU-Prolog*** компиляторы екілік кодасында жұмыс істейді. Бұл зерттеулер 80 жылдардың соңынан басталып, қазір ол білім, ғылым саласында, коммерциялық қолданбаларда кеңінен пайдаланылуда. Ресейде Прологтың «Пролог Д» (авторы С.Григорьев), «Акторный Пролог» (авторы А.Морозов), «Флэнг» (авторлары А.Манцивода, В.Петухин) деп аталатын нұсқалары пайда болды. Прологтың «Акторный Пролог» деп аталатын нұсқасы Интернет агенттерін құру және зерттеу саласында қолданылады. Бұл саланың негізгі атқаратын міндеті – нақты пайдаланушы мүддесіне сәйкес Интернеттегі деректерді жинап, талдап, осы жұмысты автоматтандыру. Яғни бұл агенттер жан-жақты іздеу жүйелері сияқты қызметті орындайды. «Акторный Пролог» нұсқасы Пролог тілінің негізгі жақсы ерекшелігін пайдаланады, яғни тілдің негізгі идеясы Интернет тақырыптарының гипермәтіндік құрылымына және адам баласының ақпаратты іздеудегі дағдысына сәйкес келеді. Прологты осы салада қолдану оның негізгі ерекшеліктері мен қасиеттерін кеңінен пайдалану болып табылады. Өйткені Пролог тілі жан-жақты программалау тілі болумен қатар, сұрату тілі және мәтіндерді синтаксистік талдауда қуатты құрал болып табылады.

Қазіргі кезде дүниежүзілік Интернет дамуында таратылатын деректер семантикасына және оны мағлұмдамалық[1] әдістермен өңдеуге деген қызығушылық болып отыр. Ол қазір ***Web 2.0*** деген атауға сәйкес дамытылуда. Осыған байланысты ***OWL*** тілінің тұжырымдамасы пайда болды. Бұл тілді классикалық Пролог тілінің кеңейтуі деп те карауға болады. Яғни оның негізінде бірінші дәрежелі логика тұжырымдамасы жатыр. Бұл ***OWL (Web Ontology Language)*** тілі

[1] Декларативті (мағлұмдамалық) тілдер – есептің қадамдардан тұратын алгоритмі емес, оның сипатын беретін программалау тәсілі.

Интертнетке арналған онтология тілі деп те аталады. Оның негізгі элементтері ***XML / Web*** стандарты бойынша қалыптасқан. Ол ***Web*** құжаттары мен қолданбаларындағы сияқты топтар мен олардың арасындағы қатынастарды сипаттайтын байланыстарды көрсетуге арналған. Мұндай тіл негізінде деректердің нысан[1] және оның қасиетін көрсететін қалыбы жатыр. Бұл тілді құрушылардың пікірлері бойынша ол ***Web*** парақтарды ғана емес, дүниедегі бар болатын басқа да нысандарды сипаттауға жарайды.

Сонымен, қазіргі таңда жаңа компьютерлік архитектуралардың және жаңа ақпараттық технологиялар бағыттарының дамуына байланысты дүние жүзінде логикалық программалаудың негізін құратын Пролог тілінің көптеген жергілікті тіл ерекшеліктеріне қызығушылықтар бар.

1.2. Пролог тілінің синтаксисі мен семантикасы

Бұл параграфта Пролог тілінің негізгі ерекшеліктері, ондағы қатынас түрлері туралы түсініктер беріледі. Пролог интерпретаторының жұмыс істеу алгоритмі келтіріледі. Прологтағы құрамдас термдер, қайталау механизмі, ішкі және сыртқы мақсаттар каралады [5].

Пролог – логикалық бағдарламалаудың (logic programming language) ең көп мәлім болған мысалы. Логикалық бағдарлама дегеніміз – логика шеңберіндегі спецификациялар жиынтығы. Пролог 1-ші дәрежедегі предикаттар теориясына негізделген. Бұл тілдің атының өзі былайша таратылады: Programming in Logic (Логикадағы Программалау). Бағдарлама орындалғанда интерпретатор үнемі логикалық ерекшеліктер негізінде шешімді шығарып отырады. 1-ші дәрежедегі предикаттар теориясы идеясын пайдалануды компьютерлік ғылымдар үшін және жасанды зерде мәселелерін шешу үшін қолданатын Пролог тілінің ең үлкен жетістігі деп білуге болады. 1-ші дәрежедегі предикаттар теориясын программалау тілінде қолдану түсінікті, көркем синтаксис пен жақсы анықталған семантиканы пайдалануға мүмкіндік береді [2].

Пролог тілінің даму тарихы теоремаларды дәлелдеу, оның ішінде тұжырымдық терістеу алгоритмін жасау кезеңінен бастау алады. Бұл алгоритмде *резолюция* (resolution) деген атқа ие болып, кейіннен Про-

---

[1] Программалық объект (нысан) – компьютер программасы өңдейтін деректер элементі. Компьютерлерге қатысты әдебиеттерде «объект» термині қолданылады.

лог тілінің есептеулеріндегі негізгі әдіс болып кеткен дәлелдеу процедурасы келтірілген.

Осындай қасиеттерге ие болған Пролог өзін программалаудың мынандай: коданы автоматты түрде генерациялау, программаларды анықтау және жоғары деңгейлі программалау тілдерін құру мәселелерін шешуге болатын құрал есебінде көрсете білді. Пролог логикаға негізделген басқа тілдер сияқты программалаудың мағлұмдамалық стилінде орындалады, яғни шектелген салалық мәселе есебін жоғары деңгейлік терминдерде конструкциялауды іске асырады. Процедуралық программалау стилінде алгоритмді орындаудың рет-ретімен жазылған нұсқаулар түріндегі программа жазылады. Логикалық программалауда компьютерге «шын деген не» екені хабарланады, ал процедуралық программалауда «қалай орындау керек» екені хабарланады. Бұндай мүмкіндік программалаушы адамға есепті шешу және сала мәселесіндегі ерекшеліктерді құруға көңіл бөлуін қамтамасыз етеді. Процедуралық бағдарламалау стилінде ол үшін «ары қарай не істеу керек» деген сұрақтың алгоритмдік нұсқаулары көрсетіліп кетеді.

Логикалық программалау принциптерін іске асыратын бағдарламалық тіл ретінде Пролог тілі ЖЗ есептерін шешуде зор рөл атқарады. Соның ішінде ең маңыздысы *декларативтік семантика* (declarative semantics). Бұл ЖЗ есептерінің өзара байланысын бірден анықтайтын құрал болумен бірге, үндестіруді орындайтын ішкі құрал және іздеу мен өзара салыстыруды іске асыратын жол болып табылады.

Фактілер мен ережелерді пайдалану. Пролог тілінің көптеген жергілікті тіл ерекшеліктері бар. Қазіргі кезде тілдің Уоррен (Warren) және Перейра (Pereira) жасаған нұсқасы қолданылады. Алайда Пролог тіліндегі деректерді қарапайым түрде бейнелеу үшін предикаттар логикасында қабылданған көптеген анықтамалар пайдаланылады. Дегенмен предикаттар логикасы синтаксисінің Пролог тілінен айырмашылығы да бар. Мысалы, Пролог тіліндегі «:–» таңбасы 1-дәрежелі предикаттар логикасының «+» таңбасына сәйкес келеді. Пролог синтаксисінің басқа да айырмашылықтарын атап өтейік. Мысалы, предикаттар логикасындағы **«НЕМЕСЕ», «Егер болмаса»** деп аталатын операция атаулары Пролог тілінде ***not*** деп белгіленеді. Пролог тілінде предикаттар аттары мен өзара байласқан айнымалылар аттары әріптен басталған әріптік-сандық таңбадан тұрады. Айнымалылар үлкен әріптен басталған әріптік-сандык таңбадан тұрады. Мәселен, likes (X, Asem) өрнегі немесе likes (Ernar, Asem) өрнегі мына фактіні білдіреді: «Әсемді әркім жақсы көреді». Немесе likes (Danjar,

Y), likes (Asem, Y) деген өрнек Даниярды және Әсемді сүйетін адамдар жиынын білдіреді. Пролог тілінде мына қатынастарды өрнектеу қажет болсын дейік: «Данияр Райханды сүйеді және Данияр Әсемді сүйеді». Оны мына түрде жазуға болады likes (Danjar, Raihan), likes (Danjar, Asem). Осылайша «Данияр Райханды сүйеді» және «Данияр Әсемді сүйеді» деген фактіні Пролог тілінде: likes (Danjar, Raihan); likes (Danjar, Asem) деп өрнектеуге болады. Ал мына: «Егер Данияр Райханды сүймесе, онда Данияр Әсемді сүйеді», – дегенді Пролог тілінде: Likes (Danjar, Asem) – not (likes (Danjar, Raihan)) деп жазуға болады. Бұл мысалдарда предикаттар логикасындағы *немесе – және – жоқ* байланыстары Пролог тілінде қалай бейнеленуі көрсетілген. Предикаттар аттары, мысалы, (likes) параметрлердің орналасу тәртібі мен мөлшері және параметрлердің тұрақты немесе өзгеріп тұратын саны нақты есептің талаптарымен (анық емес «семантика») анықталады. Пролог тілінде басқа ешқандай шектеулер қойылмайды. Қойылатын жалғыз талап: формулаларды дұрыс құру [4].

Пролог тілінде программа құру дегеніміз – ол 1-дәрежелі предикаттар логикасында салалық мәселе аумағына қатысты нысандар мен олардың арасындағы қатынастарды сипаттайтын ерекшеліктер жиынтығын жасау. Нақты есепке қатысты өзгешеліктер жиынтығы Дерекқор (ДҚ) деп аталады. Пролог интерпретаторы осы өзгешеліктер жиынтығына қатысты сұрақтарға жауап береді. ДҚ сұратуы – Дерекқордағы жазбалар сияқты логикалық синтаксисте берілген шаблондар. Пролог интерпретаторы шаблондар негізінде сұратуларды ДҚ мазмұнының логикалық себебін анықтауды жүргізеді. Пролог интерпретаторы сұратуларды былайша өңдейді: ДҚ-дағы іздеуді тереңге бойлай солдан оңға қарай жүргізеді және көрсетілген сұрату ДҚ-дағы өзгешеліктер жиынтығының логикалық себебін анықтайды. Пролог, негізінен, – түсіндірмеге жататын тіл. Тілдің кейбір нұсқалары толығымен түсіндірме режимінде жұмыс істейді. Ал кейбір нұсқаларын программа орындалуын жеделдету үшін немесе өзгешеліктер жиынтығының бөліктерін немесе барлығын компиляциялауға қолдануға болады. Пролог тілі – интерактивті тіл, пайдаланушы экранға мына таңба шыққанда «?-» сұратуды енгізуі қажет.

Мысалы, Джордж, Кейт және Сьюзидің «симпатия және антипатия әлемі» деген сипаттамасын қарастырайық. Бұл есептің ДҚ-да мынандай предикаттар жиынтығы бар [5].

likes (george, kate).

likes (george, susie). likes (george, wine). likes (susie, wine). likes (kate, gin). likes (kate, susie).

Бұл өзгешеліктер жиынтығының Джордж және оның достарының «әлемін» бейнелейтін түсіндірмесі бар. Бұл әлем ДҚ-дың үлгісі бола алады. Енді Пролог түсіндіргішіне мына сұрақтарды қоюымызға болады.

?- likes (george, kate). Yes
7- likes (kate, susie). Yes
?- likes (george, X). X = kate
X = susie X = wine
?- likes (george, beer)
no

Аталған мысалдағы бірнеше ерекше сәттерді атап өтейік. Біріншіден, likes (george, X) сұратуын өңдегенде, пайдаланушы деректерді бірінен соң бірін енгізеді, сондықтан Пролог түсіндірушісі Х-тің орнына қоятын Дереккордағы (ДҚ) барлық термдерді береді. Олар ДҚ-да қандай тәртіпте жазылса, сол тәртіпте беріледі. Алдымен kate, сосын susie және соңынан wine. Бұл процедуралық емес өзгешеліктер философиясына дөп келмесе де, анықтаушылық түрдегі тәртіп – бұл тізбектелген машиналарда іске асырылған көптеген түсіндірушілерге тән. Пролог тілінің программалаушысы ДҚ-дағы элементтерді іздеу тәртібін білуі керек. Сонымен бірге пайдаланушының «НЕМЕСЕ» деген енгізуіне де жауап беріледі. Бұл жағдайда жауапты іздеу соңғы табылған нәтижеден басталады. Келесі енгізулер сұратуға қажетті жауапты іздейді. Егер шешім болмаса, Пролог түсіндірушісі «no» деген жауап береді. Жоғарыда көрсетілген мысалдар берілетін мысал әлемдерінің тұйық (closed world assumption) екенін, яғни терістеуді (negation as failure) өтірік есебінде қабылдауға болатынын көрсетті. Пролог тілінде кез келген өрнек, егер оның терістеуінің шын екені дәлелденбесе, өтірік деп есептеледі. Пролог түсіндірушісі likes (george, beer) сұратуын өңдегенде, түсіндіруші likes (george, beer) предикатын іздейді немесе likes (george, beer) алу үшін ережені іздейді. Егер іздеу сәтсіз аяқталса, сұрату «өтірік» мәнін қабылдайды. Осылайша Пролог тілінде әлем жөніндегі барлық білім түрлері ДҚ-да сақталады деп есептеледі. Әлемнің тұйықтығы туралы жорамал тілдегі көптеген практикалық және философиялық күрделілік туғызады. Мысалы, бір фактіні ДҚ-ға енгізе алмау дегеніміз оның шын мәні туралы мағлұмат жоқ деп есептеледі. Дегенмен Әлемнің тұйықтығы туралы жорамал бұл фактінің өтірік екенін білдіреді. Егер белгілі бір предикат, мыса-

лы, likes (george, beeer) енгізілмесе немесе ол қате енгізілсе, онда оның сұрату жауабы «no» болады [9].

Жалпы, ЖЗ саласында терістеуді «өтірік» есебінде белгілеу көзқарасы өте маңызды мәселеге жатады. Бұндай белгілеу берілмеген білімдер мәселесін шешудің қарапайым тәсілі бола алады. Дегенмен білімдерді түсіндіруді қамтамасыз ететін бұдан да оңтайлы тәсілдер бар. Оларға: көпмәнді логиканы пайдалану (шын, өтірік, белгісіз) және монотонды емес пайымдаулар деп аталатын әдістік жолдарды жатқызуға болады. Жоғарыда ДҚ-да көрсетілген өрнектер Пролог тіліндегі фактілер (fact) өзгешеліктерінің мысалы бола алады. Сонымен бірге Пролог тілінде фактілер арасындағы байланыстарды сипаттайтын ережелер (rule) де бар. Ережелер логикалық импликациялар «:–.» арқылы сипатталады. Пролог тілінде ереже құрғанда «:–.» таңбасының сол жағында бір ғана предикат орналасады. Бұл предикат оң литерал (positive literal) мәнінде болуы керек, яғни ол терістеуі бар таңба болмауы қажет. Предикаттар логикасындағы импликация немесе эквиваленттік (< –, ~ және <->) қатынастары бар барлық өрнектер жоғарыда көрсетілген қалыпта болуы қажет. Бұл қалып Хорндық (Horn clause) деп аталады. Хорндық дизъюнктивтік қалыптағы импликацияның сол жағында оң мәнді жалғыз предикат орналасады. Хорндық дизъюнктивтік логикасы (Horn clause calculus) терістеу негізінде дәлелдеу жүргізетін 1-дәрежелі предикаттардың толық теориясына сәйкес болады.

Мысалы, жоғарыда келтірілген ДҚ-дағы ерекшеліктерге тағы екі досты анықтайтын ережені қосу керек болсын дейік. Бұл ереже былайша сипатталады:

friends (X, Y): – likes (X, Z), likes (Y, Z).

Бұл өрнектің түсіндірмесі мынандай: «Егер X сүйетін Z және У сүйетін Z бар болса және дүниеде Z бар болса, онда X және У – дос». Бұл жерде мыналарды атап өтуіміз керек: біріншіден, предикаттар логикасында да, Пролог тілінде де айнымалылар анықталмағандықтан, X, У және Z айнымалыларының анықталу аумағы friends ережесімен шектеледі. Екіншіден, X, У және Z, байланған немесе үндестірілген айнымалылар барлық өрнектер бойынша сәйкестендірілген. Пролог түсіндірушісі friends ережесін былайша өңдейді: алдынғы мысалдың ерекшеліктер жиынтығына friends ережесі қосылады. Пролог түсіндірушісі мынандай сұрату жасайды:

?- friends (george, susie). Yes

1.3. Пролог тіліндегі тізімдер мен арифметика

Пролог тілінде программалаудың негізгі механизмдерінің біріне рекурсия – қайтымдылық (қайта оралу) жатады. Бұл механизмдерді оқып үйренуден бұрын, тізімдерді өңдеу туралы мәселені зерттеу қажет. Тізім дегеніміз – тәртіптелген элементтер жиынын (оның ішінде тізімнің өзі де элемент бола алады) білдіретін деректер құрылымы. Рекурсия – тізімдік құрылымдарды өңдейтін табиғи әдіс. Пролог тілінде тізімдерді өңдеу үшін, үндестіру процедурасын және рекурсияны пайдаланады. Тізімдер элементтері квадрат жақшаға [] алынып, бір-бірінен үтір арқылы бөлінеді. Пролог тіліндегі тізімдердің бірнеше мысалын келтірейік [6].

[1, 2, 3, 4]
[[george, kate], [allen, amy], [don, pat]]
[tom, dick, harry, fred]
[[ ]]

Тізімнің бірінші элементі оның құйрығынан (хвост) ~ операторы арқылы бөлінеді. Тізім құйрығы деп бірінші элементті алып тастағандағы тізімді айтамыз. Мысалы, [tom, dick, harry, fred] бірінші элемент tom, ал құйрығы – тізім [dick, harry, fred]. Үндестіру шарасы және ~ операторы көмегімен тізімді құрамдас бөлшектерге бөлуге болады.

- Егер [tom, dick, harry, fred] тізімі [X ~ У] шаблонына сәйкес келсе, онда X = tom және Y = [dick, harry, fred] болады.
- Егер [tom, dick, harry, fred] тізімі [X, У ~ Z] шаблонына сәйкес келсе, онда X = tom, У = dick, Z = [harry, fred] болады.
- Егер [tom, dick, harry, fred] тізімі [X, У Z ~ W] шаблонына сәйкес келсе, онда X = tom, Y = dick, Z = harry, W = [fred] болады.
- Егер [tom, dick, harry, fred] тізімі [W, X, У, Z ~ V] шаблонына сәйкес келсе, онда W = tom, X = dick, Y = harry, Z = fred, V = [] болады.

Үндестіру шарасын тізімді құрамдас бөлшектерге бөлумен бірге тізімдік құрылымды құру үшін пайдалануға да болады. Мысалы, егер X = tom, Y = [dick] және L [X ~ Y] шаблонымен үндестірілсе, онда L айнымалысы [tom, dick] тізімімен байланыста болады. Тізімдегі бір-бірімен үтір арқылы бөлініп, тік сызыққа дейін орналасқан термдер тізім элементтері болады да, ал тік сызықтан кейін орналасқан тізім құйрығы болады.

Тізімді рекурсивті түрде өңдеудің қарпайым мысалын қарастырайық. Оны *member* предикаты көмегімен элементтің тізім элементіне

жататынын тексеру қажет. Алдымен тізімде элементтің бар-жоғын білетін предикатты анықтайық. Аталған *member* предикаты екі аргументтен тұрады: элемент және тізім. Ол егер берілген элемент тізімде бар болса, «шын» мәнін қабылдайды.

```
? member (a [a b c d, e]).
Yes
?- member(a, [1, 2, 3, 4]).
No
?- member(X, [a, b, c]).
X = a.
X = b.
X = c.
```

Енді *member* предикатын рекурсивті түрде анықтау үшін, алдымен X тізімінің бірінші элементі ме, соны анықтайық.

```
member (X, [X | T]).
```

Бұл предикат X мәні мен тізімдегі бірінші элементтің ұқсастығын тексереді. Егер ол *true* болса, онда X тізімінің бөлігінде бар ма, сол тексеріледі. Бұл тексеру былайша өтеді.

```
member (X, [Y | T]): – member (X, T).
```

Осылайша берілген элементтің тізімде бар болуы Пролог тіліндегі екі қатар предикаттар жолымен анықталады.

```
member (X, [X | T]).
member (X, [Y | T]): – member (X, T).
```

Бұл мысал тоқтау шарты рекурсивті шақыру алдында орындалатын, яғни алгоритм келесі болатын қадамның алдында орындалатын Пролог тіліндегі ішкі іздеу тәртібінің жұмыс істеу қалпын көрсетеді. Предикаттарды керісінше шақырудың тәртібінде тоқтау шарты еш уақытта тексерілмеуі мүмкін. Мысалы, member предикатының жүру жолын орындайық.

```
1: member (X, [X | T]) .
2: member (X, [Y | T]): – member (X, T) .
?- member(c, [a, b, c]) .
call 1. fail, since c ≠ a
call 2. X = c, Y = a, T = [b, c], member(c, [b, c])?
call 1. fail, since c ≠ b
call 2. X = c, Y = b, T = [c], member(c, [c])?
call 1. success, c = c
yes (to second call 2.) yes (to first call 2.)
yes
```

Пролог тілінде программалаудың жақсы стиліне көрсетілмеген айнымалыларды (anonymous variable) пайдалану жатады. Ол программалаушы мен түсіндірушіге белгілі бір айнымалылар тек шаблонға сәйкес келуді тексеруді арналғанын көрсету үшін қызмет етеді. Яғни айнымалыларды өзара байланыстыру есептеу үдерісіне кірмейді. Сондықтан X тізімінің бірінші элементімен сәйкес келетінін тексеру үшін, әдетте, member (X, [X | _]) жазбасы пайдаланылады. «_» – танбасы сұратудың унификация үдерісіндегі тізім құйрығының маңыздылығына қарамастан, құйрық мазмұны ешқандай рөл атқармайтынын көрсетеді. Рекурсивті тұжырымдар да көрсетілмеген (жасырын) айнымалыларды пайдаланады. Олар тізім басы ешқандай рөл атқармаған жағдайда, тізімде элементтің бар-жоғын тексеру үшін қолданылады. Мысалы:

```
member (X, [X_]).
member (X, [_ | T]): – member (X, T).
```

Тізімдер табиғаты мен рекурсивті басқаруды түсіну үшін, тізім элементтерін бір-бірден қатар бойынша жазуды пайдалануға болады. Мысалы, осындай жазба бойынша [a, b, c, d] тізім элементтерін жазу керек дейік. Ол үшін мына рекурсивті командаларды пайдалануға болады [7].

```
writelist ([ ]) .
writelist ([H | T]) – write (H), nl, writelist (T).
```

Бұл предикаттар тізім элементтерін әр жол қатарына бір-бірден жазады. nl – шығу ағынында жаңа жол қатарына өтуді білдіреді. Егер тізім элементтерін кері тәртіпте жазу қажет болса, онда рекурсивті предикат write командасының алдында болады. Онда тізім элементтері жазылмай тұрып-ақ тізім ең соңына дейін қаралып шығатынына кепілдік беруге болады. Содан соң тізімнің соңғы элементі жазылады да, тізімнің барлық алдыңғы элементтері үшін рекурсивті процедураны шақыру орындалады. Тізімді кері тәртіпте жазу былайша іске асады:

```
reverse_ writelist ([ ]).
reverse_ writelist ([H | T]) – reverse_ writelist (T), write (H), nl.
```

Осы предикаттардың жұмыс істеу тәртібін трассировка[1] режимінде орындап тексеріп көруге болады.

1.4. Пролог тіліндегі кесу

Пролог тіліндегі іздеуді басқаруда кесу операторын пайдалану. *Кесу (отсечение) операторы* (cut) – «!» таңбасымен беріліп, аргу-

[1] Трассировка – программаның әр командада немесе қатарда тоқтап тұрып орындалуы.

менті жоқ, мақсаттық тұжырым ретінде көрсетіледі. Оның қолданылуында бірнеше тиімділік бар. Біріншіден, бұл оператор оған қатынағанда үнемі орындалатын оператор, екіншіден, кері қайту жағдайында алдыңғы күйге келу мүмкіндігі болмаса, онда осы операторы бар барлық мақсатты тұжырымдар «өтірік» деп есептеледі. Кесу операторының қарапайым мысалын қарастырайық. Ол үшін алдымен *«Атпен жүру»* есеп мысалды қарастырайық. Шахмат ойынында ат фигурасы екі қадам көлденең бағытта, бір қадам тік перпендикуляр бағытта жүре алады. Және оның жүрісі шахмат аумағынан шығып кетпеуі керек. Жалпы жағдайда мүмкін болатын 8 жүрісі бар. Осы есептегі екі қадамдық жолды іске асыру қажет. Ол үшін келесі path2 предикатын құрайық [9].

```
path2 (X, Y) – move (X, Z) move (Z, Y).
```

Егер X және Y нүктелері арасында «аралық тоқтау Z нүктесі» болса, онда осы екі нүкте арасында екі қадамдық жол бар. Аталған мысал үшін Пролог тілінде мынандай Дерекқор бар делік.

```
move (1, 6).
move (1, 8).
move (6, 7).
move (6, 1).
move (8, 3).
move (8, 1).
```

1-ші күйден бастап екі қадамдық жол іздеу жайында сұратуға интерпретатор төрт мән береді.

```
?- path2 (1, W).
W = 7
;
W = 1
;
W = 3
;
W = 1
;
no
```

Егер path2 предикатында Кесу операторы қолданылса, онда жауап екі түрлі болады.

```
path2 (X, Y): – move (X, Z), !, move (Z, Y).
?- path2 (1, W).
```

W = 7
;
W = 1
;
no

Бұлай болатын себебі – Z айнымалысы бір ғана мән қабылдайды. Егер бірінші ішкі мақсат сәтті болса, онда Z айнымалысы 6 мәнімен байланысады да, Кесу операторы орындалады. Сондықтан ары қарай бірінші ішкі мақсатқа кері оралу болмайды және Z айнымалысы басқа мәндермен байланыспайды

Программалауда Кесу операторы бірнеше мақсатты орындайды. Біріншіден (келтірілген мысалдан анық көрінеді), программалаушы адамға іздеу ағашының қалпын басқаруға мүмкіндік береді. Егер ары қарай іздеу (толық іздеу) қажет болмаса, онда осы нүктеде ағашты кесуге болады. Осы кезде Пролог тілінің кодасы функцияны шақыруды еске салады: егер Пролог тілінің предикаты (немесе предикаттар жиыны) мәндердің (байланыстардың) бір жиынын «қайтарса» және осы кезде Кесу операторына қатынау болса, онда Түсіндіруші унификацияның басқа қойылымдары үшін іздеу жүргізбейді. Егер осы мәндер шешімді бермесе де, іздеу ары қарай жалғаспайды.

Екіншіден, Кесу операторы рекурсияны басқаруға мүмкіндік береді. Мысалы path предикатын шақырғанда:

path (Z, Z, L).

path (X, Z, L): – move (X, Y), not (member (Y, L)), path (Y, Z, [Y | L]), !.

Кесу операторын қосу мынаны білдіреді: графты[1] іздеу кезінде бір ғана шешім табылады. Шешімнің біреу болуын мына жағдай қамтамасыз етеді: басқа шешімдерді іздеу тек мына дизъюнктік өрнек path (Z, Z, L) орындалған соң ғана іске қосылады. Егер пайдаланушы басқа шешімдерді сұраса, онда path (Z, Z, L) предикаты «өтірік» мәнінде болады да, екінші path предикатын шақырады. Ол графта қайтадан іздеу үдерісін бастайды. Егер Кесу операторы рекурсивті түрдегі path предикатынан кейін орындалса, онда іздеуді ары қарай орындауға шақыру қайталанбайды.

Кесу операторын пайдаланудың тағы бір пайдасы бағдарлама жұмысын жеделдету және компьютер зердесін үнемдеу. Егер бұл опе-

[1] Граф – математикада нүктелер мен сызықтардан тұратын сызба. Нүктелер граф төбелері деп, ал сызықтар оның қабырғасы деп аталады. Графтар теориясы – көптеген сала мәселелерін талдап шешуде таптырмас сызба құрал.

ратор предикат ішінде болса, онда Кесу операторынын сол жағында орналасқан предикаттарға кері қайтуға арналған зерде сілтемелері құрылмайды. Өйткені бұл жағдайда олар еш уақытта керек болмайды. Осылайша Кесу операторын пайдалану қажетті шешімдерді табу үдерісінің зердені тиімді түрде пайдалану жағдайында өтуге мүмкіндік береді. Кесу операторы path предикатын қайтадан іске қосуға қажет рекурсияны іске қосуға және сол арқылы графтағы іздеуді жалғастыруда пайдаланылады. Осындай алгоритмдерді жүзеге асыру үшін, деректердің дерексіз типтері деп аталатын ұғымды карастырайық.

1.5. Пролог тіліндегі деректердің дерексіз типтері

Кез келген ортада тиімді программа құру үшін ақпаратты жасыру амалдары болуы және процедуралық абстракция енгізілуі керек. Графтағы іздеу алгоритмдерінде мынандай құрылымдар пайдаланылады: жиындар (set), стектер (stack), кезек (queue) және приоритетті кезек (priority 'queue). Сондықтан осы аталған құрылымдардың Пролог тілінде болуы заңды [4].

Алдында атап өткеніміздей, графтағы іздеуді орындау үшін, деректер құрылымдарын құру амалдарына: рекурсия, тізім, шаблонға сәйкестік құралдары қолданылады. Осындай құрылым блоктары көмегімен деректердің абстракты типтері[1] *(ДАТ)* жасалады. *ДАТ*-ты анықтайтын тізімдерді, рекурсивті шақыруларды өңдейтін процедуралардың барлығы осы абстракция ішінде «жасырылған». Бұл абстракциялар әдеттегі статикалық деректер құрылымынан бөлек болады [8].

Стек (stack) – қатынау мүмкіндігі тек бір жағынан ғана іске асатын сызықтық құрылым. Демек, құрылымның барлық бөліктері бір жағынан ғана қосылады немесе өшіріледі. Стекті «Last-In-First-Out» құрылымы «соңғы енді, бірінші шықты» деп атайды. Осындай құрылым мысалы есебінде тереңнен іздеу алгоритмін атап өтуге болады. Стек жұмыс істеуі үшін келесі операциялар анықталуы қажет.

1) Стекте элементтер болуын тексеру (Стек бостығын тексеру).
2) Стекке элементтерді қосу.
3) Стектегі соңғы элементті өшіру.
4) Стектегі соңғы элементті өшірмей қарау.

[1] Абстракты тип – бұл осы типке жататын элементтермен белгілі бір операцияларды немесе арнайы функцияларды орындауға мүмкіндік береді. Программада қолданылатын элементтердің барлық қасиеттері осы типтің ішіне жасырылады.

5) Стекте аталған элементтің болуын тексеру.

6) Стекке тізім элементтерін қосу.

5 және 6 операцияларын алдыңғы төрт операция негізінде анықтауға болады. Енді осы операциялардағы құрылыс блоктары тізім болатын жағдайдағы Пролог тілінде жазайық.

1) empty_stack ([ ]). Бұл предикатты Стек бостығын тексеруге немесе бос стекті құруға пайдалануға болады.
2) stack (Top, Stack, [Top | Stack]). Бұл предикат байланған айнымалылардан стек элементіне параметр есебінде берілетін стектің соңғы элементін өшіруге, қосуға және оқуға арналған операцияларды орындайды. Мысалы, егер бірінші берілген екі аргументтер байласқан айнымалылар болса, онда үшінші аргументте жаңа стек қалыптасады. Осыған ұқсас, егер үшінші элемент стекпен байланыста болса, онда стектің төбесінің мәнін алуға болады. Онда екінші аргумент соңғы элементі өшірілген жаңа стекпен байланыста болады. Сонымен бірге егер стекті үшінші аргумент есебінде жіберсек, онда стек төбесінің мәнін алуға болады.
3) member_stack (Element, Stack) – member (Element, Stack). Бұл өрнек аталған элемент стекте бар-жоғын анықтайды. Бұл нәтижені рекурсивті шақыру көмегімен де алуға болады. Ол үшін стектің келесі элементі қаралады да, ол Element аргументіне сәйкес келмесе, оны стектен шығарып тастайды. Бұл процедура стек бостығын тексеру предикаты «шын» мәніне ие болғанға дейін қайталана береді.
4) add_list_to_stack (List, Stack, Result) – append (List, Stack, Result) – жаңа Result стегін алу үшін, бірінші List аргументінің мәні екінші Stack аргументі мәніне қосылады. Осы операцияны былайша да орындауға болады. List стегінен әр элементті ығыстырып және келесі элементті уақытша стекке орналастыра отырып, List стегі бос болғанша операцияны орындау. Одан соң уақытша стектен кезек бойынша элементті ығыстырып, оны Stack стегіне уақытша стек бос болғанша қосып отыру.

Енді стекті кері тәртіпте шығаратын reverse_print_stack предикатын анықтайық. Бұл предикат стекте іздеу графындағы бастапқы күйден соңғы күйге өтетін ағымдағы жолдар кері тәртіпте сақталған жағдайда өте пайдалы болады. Бұл предикатты пайдалану жолын келесі мысалдан көруге болады [5].

reverse_print_stack (S): – empty_stack (S).

reverse_print stack (S): – stack (E, Reset, S),
reverse_print_stack (Rest),
write (E), nl.

Кезек (queue) дегеніміз – бұл деректер құрылымы. Оны әдетте (First-In-First-Out) – «бірінші кірді, бірінші шықты» құрылымы деп атайды. Бұл құрылымды да тізім есебінде қарауға болады. Ондағы элементтер бір жағынынан алынып, екінші жағынан қосылады. Кезек жайылып іздеу алгоритмінде қолданылады [9]. Кезекті іске асыру үшін келесі операциялар қажет:

1) empty_queue ([ ]). Бұл предикат кезектің бос екенін тексереді немесе жаңа бос кезекті іске қосады.
2) engueue (E, [ ], [E]). Бұл предикат былайша өрнектеледі: engueue (E, [H | T], [H | Tnew]) – engueue (E, T, Tnew). Бұл – рекурсивті предикат. Екінші аргументпен анықталған. E элементін кезекке қосады. Үшінші элемент жаңа кезекті білдіреді.
3) dequeue (E, [E | T], T). Бұл предикат жаңа кезек құрады (үшінші аргумент). Ол – екінші аргумент анықтаған бастапқы кезектегі элементті алып тастағаннан (бірінші аргумент) пайда болатын кезек.
4) degueue (E, [E | T]). Аталған кезектің келесі E элементін окитын предикат.
5) member_queue (Element, Queue) – member (Element, Queue). Бұл өрнек Element – элементі Queue кезегінде бар екенін тексереді.
6) add_list_to_queue (List, Queue, Newgueue) – append (Queue, List, New_gueue). Кезектегі барлық элементтерді тазалау.

Әрине, 5 және 6 операциялары алғашқы 4 операция негізінде іске асады.

Басымдық алған кезек. Басымдық алған кезекте (priority queue) кәдімгі кезектегі элементтер тәртіптелген және әр жаңа элемент өзіне сәйкес орынға қосылады. Кезектегі элементті өшіру операторы кезектегі ең жақсы іріктелген элементті өшіреді. Басымдық алған кезектегі іріктелген элементтің өзі кәдімгі кезек болғандықтан, оның көптеген операциялары кезек операцияларына ұқсас келеді. Бұл операцияларға мыналар жатады: empty_queue, member_queue және degueue. Соңғы операцияның элементі үшін ең жақсы іріктелген элемент алынады. Басымдығы бар кезектегі engueue операциясына insert_pq операциясы сәйкес келеді, өйткені әр жаңа элемент өзіне арналған орынға орналасуы керек.

insert_pq (State, [ ], [State]): – !.

```
insert_pq (State, [H | Tail], [State, H | Tail]): – precedes (State, H).
insert_pq (State, [H | T], [H | Tnew]): – insert_pq (State, T, Tnew).
precedes(X, Y): – X < Y. /*салыстыру операторы элемент
типтеріне байланысты*/
```

Бұндай предикаттың бірінші аргументі – жаңадан қосылатын элемент. Екінші аргумент кәдімгі бар болып тұрған басымдығы бар кезек, ал үшінші – кеңейтілген кезек. Бұл жағдайда precedes предикаты кезектегі элементтер тәртібінің орындалуына жауап береді.

Басымдық алған кезектегі келесі оператор есебінде – insert_list_pq предикаты жүреді. Бұл предикат іріктелмеген тізімді немесе элементтер жиынын басымдығы бар кезекке қосу үшін қолданады. Ол еншілес күйді басымдығы бар кезекке қосу үшін қажет болады. Осы жағдайда insert_ list_pq операторы insert_pq операторын басымдық алған кезекке әр жаңа элементті қосу үшін пайдаланады [7]. Мысалы:

```
insert_list_ pq ([ ], L, L).
insert_ list_pq ([State /Tail], L, New_ L): –
insert pq (State, L, L2),
insert_ list_ pq (Tail, L2, New_ L).
```

Жиындар. Енді деректердің дерексіз типіне жататын жиынды (set) қарастырайық. Жиын – бұл қайталанбайтын элементтер жиынтығы. Оны барлық еншілес күйлерді жинақтау үшін немесе closed тізімін сақтап тұру үшін пайдалануға болады. Элементтер жиыны, мысалы, (a, b), мынандай тізім түрінде [a, b] беріледі, яғни ондағы элементтердің орналасу тәртібі ешқандай рөл атқармайды. Жиын үшін мынандай операцияларды анықтаймыз:

empty_set, member_set, delete_if_inset және add_if_not_in_set.

Сонымен бірге жиындарды біріктіретін, салыстыратын операциялар да қажет болады: union, intersection, set_difference, subset және equal_set [7]. Мысалы:

```
empty_set ([ ]).
member set (E, S): – member (E, S).
delete_if_inset (E, [ ], [ ]).
delete_if_in_set (E, [E | T], T): – !.
delete_if_in_set (E, [H | T], [H | T_new]),
delete_if_in_set (E, T, T_ new), !.
add_if_not_in_set (X, S, S): – member (X, S), !.
add_if_not_in_set (X, S, [X | S]).
union ([ ], S, S).
```

```
union ([H | T], S, S new): – union (T, S, S2),
add_if_not_in_set (H, S 2, S_new), !.
subset ([ ], _),
subset ([H | T] S): – member set (H, S), subset (T, S).
intersection ([ ], _, [ ]),
intersection ([H | T], S, [H | S_new]): – member set (H, S),
intersection (T, S, S_ new), !.
intersection ([_ | T], S, S_ new): – intersection (T, S, S_ new), !.
set_difference ([ ], , [ ]).
set_difference ([H | T], S, T_ new): – member set (H, S),
set_difference (T, S, T_ new), !.
set_difference ([H | T], S, [H | T_new]): – set_difference (T, S, T_new), !.
equal_set (S1,S2): – subset (S1, S2),
subset (S2, S1).
```

1.6. Дерекқормен жұмыс және рекурсивті іздеу

Пролог тілінің көптеген қолданбаларында Пролог дерекқорындағы айнымалыларды пайдаланып өзгертуге, қосуға мүмкіндіктер бар. Дерекқор оны өңдейтін программалардан бөлек орналасады. Ол тұжырым түрінде берілетін белгілі бір фактілерден тұрады. Бұл фактілерді басқару ***assert*** және ***retract*** деп аталатын предикаттар көмегімен іске асады.

Пролог тілінде дерекқордың екі түрі кездеседі: статикалық, динамикалық. Әдебиетте оларды кей кезде ішкі және сыртқы дерекқорлар деп те атайды. Статикалық дерекқор программа кодаларының бір бөлігі болып табылады да, олар жұмыс уақыты кезінде өзгере алмайды. Динамикалық дерекқорды жаңа фактілермен толықтыруға немесе ондағы ескі тұжырымдарды өшіруге болады. Динамикалық дерекқордың тағы бір ерекшелігі – оны жеке файлда, белгілі бір дискіде сақтауға болады немесе оны оперативті зерделен оқуға болады.

Динамикалық дерекқор предикаттарын сипаттау үшін, Пролог тілінің ***database*** деп аталатын бөлігі болады. Мысалы:

```
database
   dstudent (string, symbol, integer).
```

Егер Пролог тілінде предикаттарға ***d*** деген латын әрпі қосылса, ол – динамикалық дерекқор предикаттарын статикалық ДҚ предикаттарынан ажыратудың басты белгісі.

Пролог тілінің Турбо-Пролог нұсқасында динамикалық деректқордағы предикаттарды өзгертуге қызмет істейтін кіріктірілген немесе ішкі (жапсырма) предикаттар жиыны бар. Оларға мыналар жатады:

asserta (X) – фактіні дерекқор басына қосады;

assertz (X) – фактіні дерекқор аяғына қосады;

retract (X) – берілген фактімен салыстырылған фактіні дерекқордан өшіреді;

consult (файл аты) – оперативті зерделе орналаскан дерекқорға фактілер қосу үшін файлды ашады. Факт ағымдағы дерекқордың соңына жазылады;

save (файл аты) – фактілерді ағымдағы дерекқорда сақтауды көрсетілген файлда іске асырады.

Енді Пролог тілінде құрылған дерекқорға жататын мысалды қарастырайық [5].

Бір қоғамдық ұйымға қатысты тұжырымдары бар дерекқор бар делік. Ақпарат осы ұйым мүшелеріне байланысты: аты-жөні, жасы, жарна мөлшері, жарна төленуі деген сияқты болып келеді. Бұл деректер мына предикатпен беріледі дейік:

```
Mushe (Familia, Vosract, Bznos).
```

Ұйым мүшелері жарналық ақшаны мына ереже бойынша төлейтін болсын:

```
Bznos (Vosrast, Tenge (1)): –
   Vozrast < 18.
Bznos (Vosrast, Tenge (2)): –
   Vozrast >= 18.
```

Дерекқорда мына фактілер бар:

```
Mushe (Қанатбаев, 33, теңге (2), төленбеген).
Mushe (Турабаев, 45, теңге (2), төленген).
Mushe (Серікова, 17, теңге (1), төленбеген).
Mushe (Алшанов, 27, теңге (2), төленген).
Mushe (Шегебаева, 17, теңге (1), төленген).
```

Енді дерекқорда жүргізілетін кейбір операциялар ережелерін анықтайық. Ұйымға кіретін жаңа мүшені ДҚ-ға енгізу үшін мына ережені пайдаланамыз:

```
Mushe_hosu (Mushe): –          /* Қосымша мүше қосу */
Assert (Mushe).
```

Ал дерекқордағы фактілер жайында ақпаратты білу үшін, мына ережені пайдаланамыз:

```
/* Мүшелер жайлы анықтаманы экранға шығару */
Aniktama_beru (Mushe (Familia, Vosract, Bznos)): –
Mushe (Familia, Vosract, Bznos),
Bznos (Vosrast, Summa),
Write (Mushe (Familia, Vosract, Summa, Bznos)),
Nl,
Fail.
Aniktama_beru ( _ ).
```

Ұйымдағы мүше туралы фактіні өшіру үшін, мына ереже іске қосылуы қажет:

```
/* Мүше жайлы ақпаратты өшіру */
Mushe_oshiru (Mushe): –
retract (Mushe),
fail.
Mushe_oshiru ( _ ).
```

Дереккорда тағы бір мынандай ереже бар делік. Бұл ереже ұйым мүшесінің жарнаны төлегені не төлемегені жайлы ақпаратты анықтайды.

```
/* Жарна төленгені немесе төленбегені жайлы мағлұмат */
Bznos_toleu (Mushe (Familia, Vosract)): –
Retract (Mushe (Familia, Vosract, tolenbegen)),
Assert (Mushe (Familia, Vosract, tolengen)).
```

Енді ұйым мүшелерінің тіркелуін қадағалайық. Яғни дереккорға жаңа мүшелерді қосамыз.

```
?- Mushe_hosu (Mushe (Қозбақова, 30, төленбеген)).
Иә
?- Mushe_hosu (Mushe (Байболов, 43, төленген)).
Иә
```

Енді экранға дереккордағы бар мүшелер тізімін шығарайық:

```
?- Aniktama_beru ( _ ).
Mushe (Қанатбаев, 33, теңге (2), төленбеген)
Mushe (Турабаев, 45, теңге (2), төленген)
Mushe (Серікова, 17, теңге (1), төленбеген)
Mushe (Алшанов, 27, теңге (2), төленген)
Mushe (Шегебаева, 17, теңге (1), төленген)
Mushe (Қозбақова, 30, теңге (2), төленбеген)
Mushe (Байболов, 43, теңге (1), төленген)
Иә.
```

Енді кімнің мүшелік жарна төлемегенін анықтайық. Ол үшін мынандай сұрату жасаймыз:

?- Bznos_toleu (Mushe (_, _, tolenbegen).

Бұл сұратуға Пролог былайша, жауап береді:

Mushe (Қанатбаев, 33, теңге (2), төленбеген)

Mushe (Серікова, 17, теңге (1), төленбеген)

Mushe (Қозбақова, 30, теңге (2), төленбеген)

Иә.

Енді осы жарна төлемегендерді тізімнен өшіреміз.

?- Mushe_oshiru (Mushe (_, _, tolenbegen)

Иә.

Енді Пролог дерекқорында кім қалғанын анықтайық:

?- Aniktama_beru (_).

Mushe (Турабаев, 45, теңге (2), төленген)

Mushe (Алшанов, 27, теңге (2), төленген)

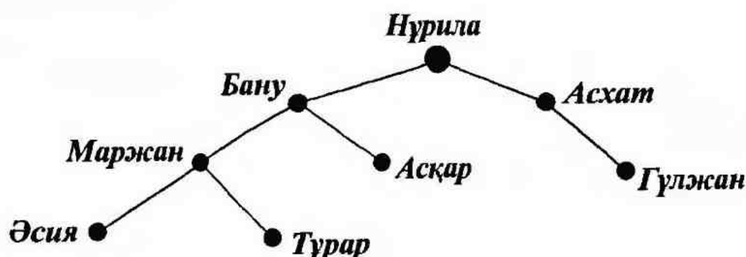
Mushe (Шегебаева, 17, теңге (1), төленген)

Mushe (Байболов, 43, теңге (1), төленген)

Иә.

Екінші бір мысалды қарастырайық. Ол 3 бөлімде келтірілген бэктрекинг тәсілімен сабақтас, яғни онда қайтару нүктесіне айналып соғатын қайтару механизмі сипатталған.

Мысал сипаты. Адамдар арасында болатын туысқандық қатынастың бір түрін қарап өтелік. Ол қатынас түрі – «әже-ана-әке-бала» қатынасы. Оның граф түріндегі сипаты 1.6.1-суретте келтірілген. Яғни Нұриланың Бану, Асхат деген екі баласынан тараған ұрпақтар деректері берілген.



1.6.1-сурет. Мысалдағы «әже-ана-әке-бала» қатынасы графы

Енді осы қатынастарға қатысты Пролог тілінде дерекқор және онымен жұмыс істейтін ережені құрамыз. Айта кететін тағы бір жай – мысалдағы жүріп өту жолдарын қысқарту мақсатында, графтағы келтірілген барлық қатынастар дерекқорға кірмеді. Мысалы (Нұрила, Бану) және (Нұрила, Асхат), (Асхат, Гүлжан) сияқты айнымалылар арасындағы қатынастар.

Мысалдың Пролог тіліндегі кодалары.
/ Дерекқордың өзі */*
mother («Маржан», «Әсия»). */* «Маржан» және «Әсия» бір-бірімен «ана» қатынасымен байланысқан */*
mother («Бану», «Маржан»). */* «Бану» «Маржанның» анасы */*
mother («Бану», «Асқар»). */* «Бану» және «Асқар» бір-бірімен «ана» қатынасымен байланысқан */*
mother («Маржан», «Тұрар»). */* «Маржан» «Тұрардың» анасы */*

/ Дерекқордың ережелер бөлігі */*
grandmother (X,Y): – */* егер белгілі бір Z айнымалысы бар болса, онда X айнымалысы Y айнымалысына әже болады, ол үшін */*
mother (X, Z), */* X айнымалысы Z айнымалысына ана, ал */*
mother (Z, Y). */* Z айнымалысы Y айнымалысына ана болуы қажет */*

Енді Пролог тілінде программаны іске қосу үшін, сыртқы мақсат есебінде барлық әжелер мен немерелердің атын атау жөнінде сұрату жасаймыз. Ол сұрату Пролог тілінде былайша жазылады:

?- (grandmother (B, V)).

Сұраққа жауап іздеуде Пролог жүйесінің жұмыс барысын қадағалау үшін, компиляторға ***trace*** директивасын береміз.

Осы ***grandmother (B, V)*** мақсаты орындалы үшін, мынандай екі кіші мақсат: ***mother (B, Z)*** және ***mother (Z, V)*** орындалуы қажет. Бірінші кіші мақсат «ана болу» қатынасын сипаттайтын бірінші сөйлеммен сәйкес келуі керек. Сондықтан ***B*** айнымалысы «Маржан» атымен сәйкестендіріледі де, ал ***Z*** айнымалысы «Әсия» атымен салыстырылады. Содан кейін екінші кіші мақсатты ***mother (Z, V)*** қанағаттандыру мәселесі қарастырылады. Бұл кезде ***Z*** айнымалысы «Әсия» мәнімен таңбаланады. Енді осы ***mother (Z, V)*** предикатындағы екінші айнымалыны іздеу сәтсіз болады, өйткені біздің дерекқорда Әсия атына байланысты, яғни оның балалары жайында мәліметтер жоқ. Енді Пролог программасы қайтадан қайту механизмі бойынша стектегі қайту

нүктесіне қайтып келеді. Сол кезде осы уақытқа дейін таңбаланып келген ***B*** және ***Z*** айнымалылары қайтадан бос болады.

Енді екінші балама қабылданады. Бұл жағдайда ***B*** айнымалысы «Бану» атына тең болып, ал ***Z*** айнымалысы «Маржан» мәнін қабылдайды.

Содан соң екінші кіші мақсатты ***mother (Z, V)*** (Z = «Маржан» мәні үшін) орындауға талпыныс жасалады. Осы ***mother*** предикатын іске асыратын процедураның бірінші сөйлемінде ағымдағы кіші мақсатпен салыстыру басталып, ***V*** айнымалысы «Әсия» мәнін қабылдайды.

Сонымен, біздің қойған сұрағымызға Пролог жүйесінің берер жауабында айнымалылар мына мәндерді қабылдайды: ***B*** = Бану, ***V*** = Әсия. Бұл жауап сұхбат терезесінде көрсетіліп, одан кейін Пролог жүйесі стекте жазылған соңғы қайту нүктесіне қайтып келеді. Сол кезде осы уақытқа дейін «Әсия» атымен таңбаланып келген ***V*** айнымалысы босайды. Кіші мақсат ***mother (Маржан, V)*** процедураның соңғы сөйлемінің тақырыбымен үндестіріледі (салыстырылады). Бұл процедура сөйлемінде ***mother*** предикаты анықталған. Осы кезде ***V*** айнымалысы «Тұрар» атымен таңбаланады. Пролог тілінің сұхбат терезесінде біздің сыртқы мақсатта қойылған сұрақтың мүмкін болатын екінші жауабы: ***B*** = Бану, ***V*** = Тұрар көрсетіледі. Енді ***mother (Маржан, V)*** кіші мақсаты үшін басқа шешім түрін іздеу мүмкіндігі жоқ.

Демек, Пролог жүйесінің трассировка терезесінде жұлдызша белгісі болмайды (* – іздеу барды білдіреді). Ал стектің қайту нүктелерінде қайту орындарын көрсететін нұсқағыштар жоқ болады. Бұл нұсқағыштар ***grandmother*** қатынасын анықтайтын ереженің екінші кіші мақсатының айнымалыға жаңа мәндерін беруі үшін қажет.

Дегенмен стектің қайту нүктелерінде Пролог программасының кейбір бөліктеріне баруды көрсететін нұсқағыштар қалып отырады. Ол – «әже болу» қатынасын анықтайтын ереженің денесіндегі бірінші кіші мақсаттың айнымалыларын таңбалау қажеттігі бар нұсқағыштар. Енді Пролог-жүйе қайтадан қайту (откат) процедурасын жасайды. Осы кезде ол өзіндегі айнымалыларды да босатады.

Бірінші кіші мақсат дерекқордағы үшінші ***mother («Бану», «Асқар»)*** деген фактімен салыстырылады. Енді біздегі кіші ***mother («Асқар», V)*** мақсатын білім қорындағы бар фактілермен салыстыру басталады. Бұл салыстыру талпынысы сәтсіз аяқталады, өйткені Пролог дерекқорында Асқардың балалары жайлы ақпарат жоқ.

Енді Пролог программасының орындалуы тағы да бірінші кіші мақсат шешімін таңдауға арналған орынға қайтадан қайтып келеді. Кіші мақсат «аналар» туралы білімдерді сипаттайтын процедураның

сөйлемімен салыстырылады. Осы кезде ***B*** айнымалысы «Маржан» атауымен нақтыланады да, ал ***Z*** айнымалысы «Тұрар» атымен таңбаланады. Бірінші кіші мақсатта салыстыруға басқа нұсқалар жоқ. Қайту нүктелері стегі бос. Трассировка терезесінде қайтуы мүмкін болатын балама шешімдер жайында хабарлайтын индикаторда ешқандай мән жоқ. Пролог жүйесі екінші кіші ***mother («Тұрар», V)*** мақсатын өз дерекқорындағы фактімен салыстыруға әрекет жасайды. Бірақ бұл талпыныс сәтсіз болады. Өйткені дерекқордағы бірде-бір фактінің бірінші айнымалысында «Тұрар» деген мән жоқ. Яғни Маржанның немерелері туралы бірде-бір дерек болмағандықтан, Пролог оны таба алмайды. Сонымен бірге Пролог жүйесі (Нұрила, Бану) және (Нұрила, Асхат), (Асхат, Гүлжан) сияқты айнымалылармен де жұмыс істеуі мүмкін. Олардың барлығы сәтсіз аяқталады, өйткені Пролог дерекқорында олар туралы бірде-бір факт көрсетілмеген.

Сонымен, Пролог программасы аяқталады. Прологтың сұхбат терезесінде жұмыс істеу кезінде табылған екі шешімнің нәтижесі көрсетіледі:

B = Бану, ***V*** = Әсия
B = Бану, ***V*** = Тұрар
1 Solutions Екі түрлі шешім бар.

Пролог тіліндегі рекурсивті іздеу. Рекурсия әдісі – салыстыру әдісіне негізделген логикалық программалауда қолданылатын негізгі ұғымдардың бірі. Бұл әдіс өзінің ең жақсы жақтарын тізімдік құрылымдарда көрсетеді. Пролог тіліндегі тізім мен оның басы, құйрығын анықтайтын процедуралар мынандай:

list ([ ]); – тізімнің өзі, head (H, [H | T]); – тізімнің басы,
list ([H | T]) ← list (T); tail (T, [H | T]). – тізім құйрығын анықтау.

Рекурсия әдістерінің өзі екі түрлі тәсілден тұрады. Оның бірі «құйрықтық» деп аталады. Немесе оны әдебиетте «ену рекурсия әдісі» деп атап жүр. Бұл тәсілде рекурсивті деректер құрылымын қалыптастыру үшін, жинақтағыш деп аталатын құрылым пайдаланылады. Оны кей кезде аккумулятор деп те атайды. Логикалық қорытынды шығарудың бірінші қадамында жинақтағышта бос тізім болады. Келесі қадамда жинақтағышқа айналымға түсетін тізімнің басы ғана түседі де, ал тізімнің өзі «бассыз» қалады. Ең соңғы қадамда айналымға түскен тізім бос тізім болады. Жинақтағышта пайда болған құрылым аргумент-нәтиже болады. Бұл «құйрықтық» рекурсияның тағы бір маңызды жағы – компьютер зердесі қайту функциясының адрестерін сақтауға ғана бөлінеді. Ал жүргізілетін есептеулер тұрақты зердеде

орындалады. Ал екінші тәсілді «құйрықсыз» деп атайды. Әдебиетте оны «шығу рекурсия әдісі» деп атап жүр. Бұл тәсілдің ерекшелігі мынада: логикалық есепті шешу кезінде деректердің рекурсивтік шығу құрылымы логикалық қорытындымен бірге құрылып отырады.

Жоғарыда қарастырылған «Атпен жүру» мысалында предикаттар теориясындағы бейнеленуін келтірейік. Ат жүрісін өлшемі 3х3 болатын келесі тақтада бейнелейміз. Пролог тіліндегі мүмкін болатын ат жүрісін move предикаты көмегімен жазуға болады. Ал path предикаты аргументтер арасындағы 0-ден көп жүрістер санын белгілейтін жол санын анықтайтын алгоритмдерді білдіреді. Ал ондағы path предикаты рекурсивті түрде анықталған [9].

```
move (1, 6). move (3, 4). move (6, 7). move (8, 3).
move (1, 8). move (3, 8). move (6, 1). move (8, 1).
move (2, 7). move (4, 3). move (7, б). move (9, 4).
move (2, 9). move (4, 9). move (7, 2). move (9, 2).

path (Z, Z).
path (X, Y): – move (X, W), not (been (W)),
assert (been (W)), path (W, Y).
```

Соңғы белгіленген path предикатының анықтамасы – оның Пролог тіліндегі іске асуы. Жоғарыда келтірілген assert – Пролог тіліндегі ішкі предикат, ол өз аргументтерін ерекшеліктер дерекқорына орналастырады және үнемі «шын» деген мәнге ие. Ал been предикаты бұрынғы болған алдыңғы күйлерді жазуға және циклдерді болдырмауға пайдаланылады. Жалпы, been предикатын бұлай пайдалану глобалды[1] айнымалыларды қолданбай, предикаттар логикасында өзгешеліктер жасау жөніндегі программа құру мақсатына қайшы келеді. Мысалы, been (3) деректер қорына қосқан соң, ол глобалды мағынаға ие факт бола алады. Бұл фактіні осы деректер қорындағы кез келген процедура қолдана алады.

Программаны басқаруда глобалды құрылымдар жасау өнімдік жүйелерде есеп логикасы (есеп ерекшелігі) программа басқарудан бөлек сақталатын жүйелерді үлгілейтін негізгі принципіне қайшы келеді. Өнім жоғарыдағы мысалда been құрылымымен программа орындалуын жаңартуға арналған глобалды ерекшелік ретінде құрылған. Яғни path предикатын шақырғанда жүріп өткен күйлерді есепке алу үшін және циклдарды болдырмау үшін тізімді пайдалануға болады. Ұқсас

[1] Глобалды айнымалылар – функционалдық және логикалық программалауда айнымалы аты оның мәні және орнын (мекенін) анықтайды.

күйлерді (циклдарды) анықтау үшін member предикатын пайдалануға болады. Бұндай тәсіл been (W) – глобалды тұжырымын қолдану мәселесін айналып өтуге мүмкіндік береді.

Пролог тіліндегі тереңнен іздеу әдісінің қайталау алгоритмі көмегімен шешімі былайша іске асады:

```
path (Z, Z, L).
path (X, У, L): – move (X, Z),
not (member (Z, L)),
path (Z, Y, [Z | L]).
```

мұндағы member – жоғарыда анықталған предикат.

Осы path предикатының үшінші аргументі жүріп өткен күйлерден тұратын тізімді көрсететін жергілікті айнымалы. Жаңа күй генерацияланғанда (move предикатын пайдалана отырып), ол [Z (L)] күйлер тізімінің басында болады да, қайтадан path шақырады. Бұл шақыру егер осы уақытқа дейін айнымалы жүріп өткен күйлер тізімінде болмаса, not (member (Z, L)) болады, ал path предикатының барлық параметрлері жергілікті болады және олардың ағымдағы мәні іздеу графын шақыру орнына тәуелді. Әр рекурсивтік шақыруда бұл тізімге жаңа күй қосылады. Егер берілген күйдің туындылары сәтсіз болса, path шақыруы да сәтсіз болады. Егер түсіндіруші аталық шақыруға қайтса, онда жүріп өткен күйлерден тұратын тізімді беретін үшінші параметр өзінің алдыңғы мәнін қабылдайды.

Осылайша графтағы қайталап іздеу үдерісінде күйлер тізімге қосылып және өшіріліп отырады. Ал path шақыруы сәтті аяқталғанда, алғашқы екі параметр бірден мән қабылдайды. Үшінші параметр – шешім жолындағы жүріп өткен күйлерден тұратын тізім. Бұл жолда аталған күйлер кері тәртіпте беріледі. Осылайша шешім табудың барлық кезеңдерін басып шығаруға болады. Сонымен, тізімдерді және графтағы қайталап іздеу көмегімен «Ат жүрісі» есебіне арналған Пролог өзгешелігін path және move, member предикаттарын пайдаланып құруға болады.

Ал path (X, Y, [X]) командасына арналған Пролог түсіндірушісін шақыру, мұндағы X және Y – 1-ден 9 саны аралығындағы сандар. Ол X күйінен Y күйіне көшетін жолды табуға мүмкіндік береді. Егер, әрине, мұндай жол болса. Үшінші параметр X алғашқы күйінің жүріп өткен күйлерден тұратын тізімін іске қосады. Пролог тілінде таңбалар регистрі есепке алынбайды. Алғашқы екі параметр есептің анықталу аумағындағы кез келген күйлерді бейнелейді, ал үшінші параметр күйлер тізімін көрсетеді. Үндестіру көмегімен барлық мүмкін болатын

деректер типтерінің шаблондарға сәйкестігінің жалпылама тексерілуі орындалады.

Осылайша path – кез келген графқа қолдануға болатын тереңнен іздеудің жалпылама алгоритмі. 3X3 тақтасында берілген «ат жүрісі»есебіне қайтадан оралайық. Осы мысалдың 3х8 тақтасына арналған түрінің Пролог тіліндегі шешімін қарастырайық. Ол үшін path алгоритмінің екі жағын нөмірлейік [9].

1) is path (Z, Z, L).
2) is path (X, Y, L): - move (X, Z), not (member (Z, L)), path (Z, Y, [Z (L]).
?- path (1, 3, [1]).

Мысалдың жүріп өту жолы мынадай болады.

path (1, 3, [1]) attempts to match 1. fail 1 ≠ 3.
path (1, 3, [1]) matches 2, X is 1, Y is 3, L is [1]
move (1, Z) matches Z as 6, not (member (6, [1])) is true,
call path (6, 3, [6, 1])
path (6, 3, [6, 1]) attempts to match 1. fail 6 ≠ 3.
path (6, 3, [6, 1]) matches 2. X is 6, Y is 3, L is [6,1].
move (6, Z) matches Z as 7, not (member (7, [6, 1])) is true,
path (7, 3, [7, 6, 1]).
path (7, 3, [7, 6, 1]) attempts to match 1. fail 7 ≠ 3.
path (7, 3, [7, 6, 1]) matches 2. X is 7, Y is 3, L is [7, 6, 1].
move (7, Z) is E = 6, not (member (6, [7, 6, 1])) fails, backtrack!.
move (7, Z) is Z = 2, not (member (2, [7, 6, 1])) true,
path (2, 3, [2, 7, 6, 1]), path call attempts 1, fail, 2 ≠ 3.
path matches 2, X is 2, Y is 3, L is [2, 7 ,6, 1].
move matches Z as 7, not (member (...)) fails, backtrack!.
move matches Z as 9, not (member (...)) true,
path (9, 3, [9, 2, 7, 6, 1]); path fails 1, 9 ≠ 3.
path matches 2, X is 9, Y is 3, L is [9, 2, 7, 6, 1].
move is Z = 4, not (member (...)) true,
path (4, 3, [4, 9, 2, 7, 6, 1]); path fails 1, 4 ≠ 3.
path matches 2, X is 4, Y is 3, L is [4, 9, 2, 7, 6, 1]
move Z = 3, not (member ( )) true,
path (3, 3, [3, 4, 9, 2, 7, 6, 1]).
path attempts 1, true, 3 = 3, yes

yes

yes

yes

yes

yes

yes

Қорытындылай келе, мынаны атап өтелік. Path – рекурсивтік шақыруы графтағы іздеуге арналған жалпы басқару құрылымы немесе қабықша (shell). path (X, Y, L) X – ағымдағы күйі, Y – мақсатты күйі. Егер X және Y дәл келсе, рекурсия тоқталады. L – X баратын ағымдағы жол бойындағы күйлер тізімі. Z – жаңа күйі табылғанда move (X, Z) шақыруы көмегімен, бұл күй [Z | L] тізіміне орналасады. Тізімдегі күйдің бар болуы not (member (Z, L)) шақыруы арқылы тексеріледі. Бұл тексеру табылған жолда циклдер болмауына кепілдік береді. Жоғарыда аталған жолды іздеу алгоритміндегі күйлердің L тізімі мен closed тұйық жиынының арасындағы айырмашылық мынадай: жиын барлық жүріп өткен күйлерді қамтиды, ал L тізімінде тек ағымдағы жол болады. Айырмашылықты жою үшін path шақыруы кезіндегі жазбаны кеңейтіп, онда барлық жүріп өткен күйлерін еске сақтау керек.

1.7. Пролог тіліндегі жаттығулар

Біз жаттығуларды Пролог жүйесінің Турбо-Пролог нұсқасында қарастырамыз. Бұл тілдің өз компиляторы бар. Командалық жолға PROLOG командасын енгізу арқылы Турбо-Пролог жүйесін іске қосасыз. Экранға Турбо-Пролог тілінің «Бас меню» деп аталатын терезесі шығады. Бұл терезенің жоғарғы қатарында пайдаланушыға қажетті жеті түрлі опциялар (командалар) бар. Олардың қызметі келесі түрде болады:

1. Программаны есептеуді іске қосады (Run);
2. Программаны трансляциялайды (Compile);
3. Программа мәтінін жөндейді (Edit);
4. Компилятор опцияларын анықтайды (Options);
5. Файлдармен жұмыс істейді (Files);
6. Жеке қажеттіліктерді баптауды орындайды (Setup);
7. Жүйеден шығады (Quit).

Бұл командалар екі түрлі тәсілмен жұмыс атқарады:

- таңдап алынған команданың бірінші әрпіне сәйкес, мысалы (R, C, E, O, F, S, Q) деген сияқты;
- меню ішінде мына «→» және «←» бағдаршалар мен Enter пернесінің көмегімен.

Командамен жұмыс аяқталғанда одан шығу үшін Esc пернесін басса жеткілікті. Бас терезеде сонымен бірге төрт қосымша ішкі терезе бар. Сол жақ бұрышта Турбо-Прологтың Editor – редактор терезесі орналасқан. Оң жағында Dialog сұхбат терезесі болса, сол жақ төменгі

бұрышта Messege – хабарларды экранға шығаратын терезе бар. Ал оң жақ төменгі бұрышта программаның қадамдық орындалуын қамтамасыз ететін Trace терезесі орналасқан. Редактор терезесінің жоғарғы қатарында көрінетін ақпарат түрлері мыналарды білдіреді: Indent қатар жолдарды автоматты түрде жөндеу режимі іске қосылған деген мәліметті берсе, ал Insert катарларға таңбаларды кіріктіруге болады деген ақпаратты береді. Турбо-Прологтың жұмыс файлының үнсіз келісім бойынша атауы WORK.PRO деп келсе, жалпы жағдайда Пролог программасының кеңейтуі *.PRO деп жазылады.

Турбо-Прологта жазылған программа бес бөлімнен тұрады. Оның ең маңызды сөздері: ***domains, database, predicates, goal, clauses*** түрінде болып, сол арқылы бөлімнің басталуын хабарлайды. Программалардың барлығында осы бес бөлімнің болуы міндетті емес. Сонымен бірге Турбо-Пролог тілінде түсіндірмелер (комментарий) программаның кез келген жерінде кез келген ұзындықтағы мына таңбамен: /* және */ белгіленеді. Бұл бес бөлім түрлері мыналарды бейнелейді:

domains – деректер типтерін анықтайды;

database – динамикалық дерекқордың предикаттарын анықтайды;

predicates – предикаттардың өздерін анықтайды;

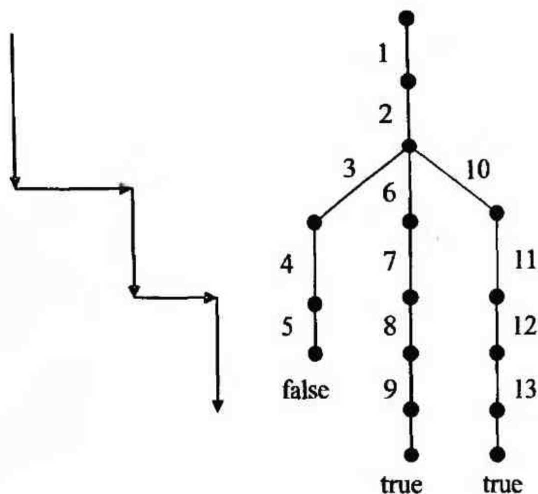
goal – ішкі мақсатты анықтайды;

clauses – фактілер, ережелер, яғни программаның мәтіні жазылады.

Жалпы жағдайда Турбо-Пролог тіліндегі кез келген программада ***predicates*** және ***clauses*** бөлімдерінің болуы міндетті, ал қалғандарының болуы-болмауы міндетті емес. Егер программада ***goal*** бөлімі болса, онда ондай программаларды ішкі мақсатты программалар деп атайды. Егер бұл бөлім болмаса, онда сыртқы мақсатты программа деп атайды. Сыртқы мақсатты программада пролог жүйесі сұхбат терезесіне сізді міндетті түрде шақырады. Турбо-Пролог тілі – интерактивті немесе сұхбаттық тіл.

Программа орындалуы мақсат операторы табылғаннан бастап іске асады. Мақсат – ол программа шешуге тиісті мәселенің қалыптастырылған түрі. Ішкі мақсаттарды Пролог өзі тапса, ал сыртқы мақсатты ол пайдаланушыдан сұрайды. Бұл жағдайда Турбо-Пролог экранға Goal (мақсат) деген шақыруды шығарады. Содан кейін Турбо-Пролог берілген мақсатты программадағы бар фактілер мен ережелермен салыстырады. Салыстыру принципінің сызбанұсқасын өз есіңізде мықты сақтауыңыз қажет. Ол мынандай болып келеді: ***«жоғарыдан төмен қарай және солдан оңға қарай жүру»***. Бұл принцип біз 3-бөлімде қарастырған тереңнен іздеу алгоритміне келеді. Бұл принципті тағы

да еске сақтау мақсатында [14] келтірілген логикалық программаның мақсатын есептеуге арналған шешімдерді іздеу ағашының графын келтіре кетейік (1.7.1 суретті қара).



1.7.1-сурет. Шешімдерді іздеу ағашының графы

Суреттегі граф доғалары ***«жоғарыдан төмен қарай және солдан оңға қарай жүру»*** стратегиясы бойынша нөмірленген. Оны суреттің сол жағындағы бағдаршалар бағыты көрсетеді. Егер мақсат фактіге сай келсе, онда Турбо-Пролог True (Шын) деген жауапты береді де, керісінше болса, False (Өтірік) немесе No деп жауап береді. Егер мақсатта айнымалылар болса, онда Турбо-Пролог олардың шешім беретін мәндерін келтіреді немесе No solutions (шешім жоқ) деген хабарды береді.

Турбо-Пролог тілінде программа құрғанда, мына тәртіпті сақтау қажет:

– тілдегі нысандар мен қатынастардың барлығы кіші әріптермен жазылады;

– ең алдымен қатынастар атаулары (предикат) жазылады. Содан соң үтір арқылы нысандар атаулары жазылып, нысандар атауларының тізімі дөңгелек жақшаға алынады;

– әрбір факт, мақсат соңында міндетті түрде нүкте қойылады.

Алдыңғы параграфта атап өткеніміздей, Прологтағы дерекқорлар екі түрлі: статикалық және динамикалық болады. Ол Прологтың ***database***

деген бөлімінде орналасады. Тағы сонда көрсетілгендей, Дерекқордағы өзгерістерді мына ішкі предикаттар іске асырады: asserta (X), ssertz (X) деректерді қордың басына немесе соңына жазады; retract (X) Дерекқордағы фактілерді өшіреді; consult (файл аты) оперативті зердедегі файлды ашады.

Прологтағы жаттығулар. *1-мысал. Қайталап іздеу.*

Белгілі бір деректер құрылымында іздеуді іске асыратын Пролог тілінде программа жазу. Әрбір фактіде ең аз дегенде үш элемент болуы керек. Деректерді реляциялық түрде беруге болады. Онда кестедегі жазба саны 15-тен кем болмауы қажет. Осы құрылымдағы іздеуді жазбаның бір немесе екі атрибуты бойынша құру қажет. Мысалда мемлекеттер жайлы ақпарттың кесте түріндегі құрылымы берілген.

Мемлекет	Аумағы		Халқының саны		Астанасы
	мың шақ.^{2}	орны	мың адам	орны	
1	2	3	4	5	6
Австралия	7 686,8	6	21 585,1	52	Канберра
Франция	674,8	47	64 473,1	20	Париж
Үндістан	3 287,6	7	1 131 191,1	2	Нью-Дели
Венгрия	93,0	109	9 930,9	79	Будапешт
Канада	9 984,7	2	33 091,2	37	Оттава
Қытай	9 570	3	1 322 178,2	1	Пекин
Ресей	17 075,4	1	141 887,5	9	Мәскеу
АҚШ	9 518,9	4	304 000,0	3	Вашингтон
Англия	244,8	76	60 776,2	21	Лондон
Грекия	131,9	94	10 964,0	70	Афины
Қазақстан	2 724,9	9	15 658,3	61	Астана
Мадагаскар	587,0	45	19 448,8	55	Антананариву
Мальдивы	0,3	186	298,9	166	Мале
Эстония	45,0	129	1 342,4	151	Таллин
Жапония	377,8	60	127 433,5	10	Токио
Чехия	78,9	114	10 403,1	79	Прага

Енді осы есептің Пролог тіліндегі кодаларын келтірейік. Мысалдар [38] алынды.

```
/* Дерекқордың деректер типтерін анықтау бөлігі */
DOMAINS
   territory = ter (real, integer).
   population = pop (real, integer).
   info = c (string, territory, population, string).
/* Дерекқордың предикаттар бөлігі */
PREDICATES
   country (info).
     show.              /* кестені экранға шығару */
     search (string).
   search (integer, integer).
/* Дерекқордың фактілері мен ережелер бөлігі */
CLAUSES
   country (c («Australia», ter (7686.8, 6), pop (21585.1, 52), «Kanberra»)).
   country (c («France», ter (674.8, 47), pop (64473.1, 20), «Paris»)).
   country (c («India», ter (3287.6, 7), pop (1131191, 2), «New Delhi»)).
   country (c («Hungary», ter (93.0, 109), pop (9930.9, 79), «Budapest»)).
   country (c («Canadian», ter (9984.7, 2), pop (33091.2, 37), «Ottawa»)).
   country (c («China», ter (9570, 3), pop (1322178.2, 1), «Pekin»)).
   country (c («Russia», ter (17075.4, 1), pop (141887.5, 9), «Moskow»)).
   country (c («USA», ter (9518.9, 4), pop (304000.0, 3), «Washington»)).
   country (c («BreatBritain», ter (244.8, 76), pop (60776.2, 21), «London»)).
   country (c («Greece», ter (131.9, 94), pop (10964.0, 70), «Athenes»)).
   country (c («Kazakhstan», ter (2724.9, 9), pop (15658.3, 61), «Astana»)).
   country (c («Madagascar», ter (587.0, 45), pop (19448.8, 55), «Antananarivo»)).
   country (c («Maldives», ter (0.3, 186), pop (298.9, 166), «Male»)).
   country (c («Estonia», ter (45.0, 129), pop (1342.4, 151), «Tallinn»)).
   country (c («Japan», ter (377.8, 60), pop (127433.5, 10), «Tokyo»)).
   country (c («Czechia», ter (78.9, 114), pop (10403.1, 79), «Prague»)).

/* Экранға шығару ережесі */
   show:-write («****************************************\n»),
   write («* COUNTRY\t* TERRITORY \t* POPULATION \
   t* CAPITAL *»),   nl, write («* \t* km place \t* people place\
   t* \t*\n»),
   write («********************************************\n»), nl,
   country (c (X, ter (At, Bt), pop (Ap, Bp), Y)),
   writef («%s\t\t%-1\t%u\t%-1\t%u\t%s\n», X, At, Bt, Ap, Bp, Y), fail.
```

```
/* Бір атрибутты іздеу ережесі */
    search (X):-
    country (c (Y, ter (_, _), pop (_, _), X)),
    writef («The %s is the capital of the %s.\n», X, Y), fail;
    country (c (X, ter (At, Bt), pop (Ap, Bp), Y)),
    writef («The capital of %s is %s.\nThe territory is %-l, zanimaet
    %u mesto v mire.\nThe population is %-l, zanimaet %u mesto v
    mire.\n», X, Y, At, Bt, Ap, Bp), fail.

/* Негізгі екі атрибутты ереже */
    search (T, P): –
    country (c (X, ter (_, Bt), pop (_, Bp), _)),
    Bt <= T, Bp <= P,
    writef (« %s, «, X),
    fail.
```

Нәтижелер. Шығу мәліметтерін экранға шығару. Прологта ақпаратты экранға шығару ***show*** командасы көмегімен іске асады. Мына 1.7.2-суретте берілген кестенің өзін экранға шығардық.

```
D:\55DF~1\82ACE~1\8233~1\TURBOP~1\TURBOP~1\PROLOG.EXE
********************************************************************
* COUNTRY          *  TERRITORY      *  POPULATION       * CAPITAL  *
*                  * km     place    * people   place    *          *
********************************************************************

Australia                 7686.8   6         21585.1 52       Kanberra
France            624.8   47       64473.1   20       Paris
India             1287.6  7        1131191   2        New Delhi
Hungary           93      109      9930.9    79       Budapest
Canadian                  9984.7   2         33091.2 37       Ottawa
China             9570    3        1322178.2          1       Pekin
Russia            17075.4 1        141887.5           9       Moskwa
USA               9518.9  4        304000    3        Washington
GreatBritain              244.8    76        60776.2 21       London
Greece            131.9   94       10964     70       Athenes
Kazakhstan                2724.9   9         15658.3 61       Astana
Madagascar                587      45        19448.8 55       Antananarivo
Maldives                  0.3      186       298.9   166      Male
Estonia           45      129      1342.4    151      Tallinn
Japan             377.8   60       127433.5           10       Tokyo
Czechia           78.9    114      10403.1   79       Prague
No
Goal:
```

1.7.2-сурет. Деректерді экранда көрсету

Бір атрибут бойынша іздеу. Іздеуді бір атрибут бойынша орындау. Мысалы, мемлекет аты және қала бойынша іздеу (1.7.3-суретті қара). Біз бір атрибутпен іздеуде программа сұратуында ***goal: search («China»)*** деп көрсеткен болатынбыз. Яғни іздеу айнымалысына нақты мән беру

арқылы бір атрибутты іздеу ережесі бойынша іздедік. Мысалы, Қытай (China) мемлекеті және Мәскеу (Moskow) қаласы жайлы сұратулар экранға қажетті ақпаратты шығарады:

1.7.3-сурет. *Мемлекет аты және қала бойынша іздеу нәтижесі*

Екі атрибут бойынша іздеу. Аумағы және халқының саны жағынан 10-орында тұрған елдер мен аумағы бойынша 100-орыннан жоғары тұрған және халқының саны бойынша 50-орыннан жоғары тұрған мемлекеттерді экранға шығару керек. Осы жерде мынаны айта кетуіміз қажет. Бірінші атрибут сұратуының түрі мынандай болып келеді: ***goal: search (10, 10).*** Яғни программадағы негізгі ереженің тақырыбындағы екі атрибуттың екеуі де қамтылған.

1.7.4-сурет. *Екі атрибут бойынша іздеу нәтижесі*

2-мысал. Рекурсия. Мынандай ***cos(n)*** функцияларынан тұратын қатар берілген. Осы функция сандарының қосындысын есептейтін Пролог тілінде рекурсивті программа жазу қажет. Нәтижені кесте түрінде шығару керек. Рекурсияның «құйрықты», «құйрықсыз» деп аталатын түрлерін қамту қажет.

Жұмыстың орындалуы. ***«Құйрықсыз» рекурсия:***

```
PREDICATES                  /* предикаттар бөлігі */
   sum (integer, real).
CLAUSES                     /* фактілері мен ережелер бөлігі */
   sum (0, 1): – !.
   sum (N, R): –
   Next_N = N-1,
```

```
   sum (Next_N, P),
   R = cos(N) + P,
   writef (« % \t %-4», N, R), nl.
GOAL                          /* Мақсат бөлігі */
   write («*** Lab 3. Recurse ***»), nl, nl,
   write («Enter quantity of elements of a number:»),
   readint (X), nl,
   write («Number SumCos»), nl,
   sum (X, Res).
```

«Құйрықты» рекурсия мысалы:

```
PREDICATES                    /* предикаттар бөлігі */
   sum (integer, real, real).
CLAUSES                       /* фактілері мен ережелер бөлігі */
   sum (0, _, _): – !.
   sum (N, R, P): –
   Next_N = N-1,
   R = cos(N) + P,
   sum (Next_N, R, R),        /* жинақтаушы */
   writef (« % \t %-4», N, R), nl.
GOAL                          /* Мақсат бөлігі */
   write («*** Lab 3. Recurse ***»), nl, nl,
   write («Enter quantity of elements of a number: «),
   readint (X), nl,
   write («Number SumCos»), nl,
   sum (X, Res, 0).
```

Нәтижелер. Рекурсия әдісімен алынған нәтиженің экрандық түрі. (1.7.5-суретін қара).

```
       *** Lab 3. Recurse ***

Enter quantity of elements of a number: 5

Number        SumCos
 1          1.5403023059
 2          1.1241554693
 3          0.1341629727
 4           0.5174806481
 5          -0.2358184627
```

1.7.5-сурет. Рекурсия әдісі нәтижесінің экрандық түрі

3-мысал. Тізімдерді өңдеу. Мынандай n элементі бар тізім берілген. Осы тізімнің n-ші элементін экранға шығару керек.

```
DOMAINS                    /* деректер типтерін анықтау бөлігі */
   list = integer*.
PREDICATES                 /* предикаттар бөлігі */
   nth_element (integer, list, integer).
CLAUSES                    /* фактілері мен ережелер бөлігі */
   nth_element (1, [A | _], A): – !.
   nth_element (N, [_ | L], A): –
   N1 = N-1,
   nth_element (N1, L, A).
GOAL                       /* Мақсат бөлігі */
   write («*** Lab 4. List ***»), nl, nl,
   write («Enter number of element of the list:»),
   readint (N),            /* санды енгізу */
   nth_element (N, [0, 1, 2, 3, 4], A),
   writef («Element # % = %», N, A), nl.
```

Нәтижелер. Тізімді өңдеу мысалы нәтижесінің экрандық түрі. (1.7.6-суретін қара).

```
c:\ [□хръЄштэю LAB4.EXE]
        *** Lab 4. List ***

Enter number of element of the list: 2
Element # 2 = 1
```

1.7.6-сурет. Тізімдерді өңдеу нәтижесінің экрандық түрі

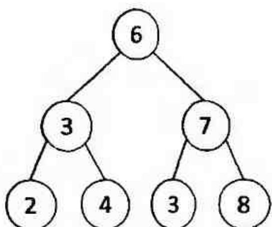
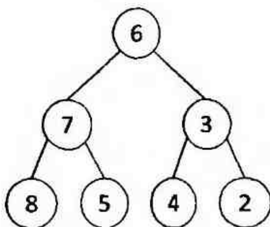
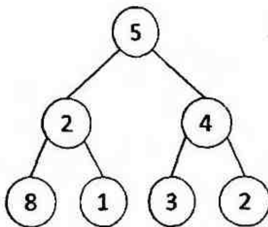
4-мысал. Деректердің рекурсивті құрылымдары (ағаштар). Берілген ағаш түрі тәртіптелгенін көбеюі немесе азаюы бойынша анықтау қажет.

```
DOMAINS                    /* деректер типтерін анықтау бөлігі */
   tree = t (integer, tree, tree); empty ()
PREDICATES                 /* предикаттар бөлігі */
   print_tree (tree).
   order (tree)
   order_left (integer, tree)
   order_right (integer, tree)
CLAUSES                    /* фактілері мен ережелер бөлігі */
```

```
print_tree (empty): – !.
print_tree (t (R, Left, Right)): –
write (R, '\t'),
print_tree (Left),
print_tree (Right).
order (empty): – !.
order (t (_, empty, empty)): – !.
order (t (R, Left, Right)): –
order_left (R, Left),
order_right (R, Right),
order (Left),
order (Right),
write («Tree order by vozrast\n»);
order_left (R, Right),
order_right (R, Left),
order (Left),
order (Right),
write («Tree order by ubivan\n»).
order_left (_, empty).
order_left (T, t (L, _, _)): – T >= L.
order_right (_,empty).
order_right (T, t (R, _, _)): – T <= R.
GOAL                                            /* Мақсат бөлігі */
write (« *** Lab 5. Tree ***»), nl, nl,
Tree1 = t (6, t (3, t (2, empty, empty),
t (4, empty, empty)),
t (7, t (3, empty, empty),
t (8, empty, empty))),
Tree2 = t (6, t (7, t (8, empty, empty),
t (5, empty, empty)),
t (3, t (4, empty, empty),
t (2, empty, empty))),
Tree3 = t (5, t (2, t (8, empty, empty),
t (1, empty, empty)),
t (4, t (3, empty, empty),
t (2, empty, empty))),
write («Tree:»),
print_tree (Tree3), nl,
order (Tree3),
```

```
write («Tree order!\n»), !;
write («Tree not order\n»).
```

Нәтижелері. Программа жұмысының нәтижесін үш ағаш мысалымен көрсетеміз. Келесі көрсетілген ағаштар көбеюі, азаюы бойынша және тәртіптелмеген түрде берілген (1.7.7-суретін қара).

Көбеюі бойынша тәртіптелген:	Азаюы бойынша тәртіптелген:	Тәртіптелмеген ағаш:
		

1.7.7-сурет. Ағаш түріндегі деректер

Нәтижелер. Көбеюі бойынша тәртіптелетін № 1 ағаш түріне арналған нәтиже (1.7.8-суретін қара).

[Ξxpъ€штэю LAB5.EXE]

1.7.8-сурет. Көбеюімен тәртіптелетін № 1 ағаштың экрандық түрі

Азаюы бойынша тәртіптелетін № 2 ағаш түріне арналған нәтиже (1.7.9-суретті қара). *Ал тәртіптелмеген № 3 ағаш түрінің экрандық түрі* 1.7.10-суретте келтірілген.

[Ξxpъ€штэю LAB5.EXE]

1.7.9-сурет. Азаюмен тәртіптелетін № 2 ағаштың экрандық түрі

1.7.10-сурет. Тәртіптелмеген № 3 ағаштың экрандық түрі

5-мысал. Дерекқор. Берілген деректерден белгілі бір шарттарға сәйкес дерекқор құру қажет. Бастапқы дерекқорда әр түрдегі фактілер бар. Пролог тілінде келесі аталған іс-әрекеттерді орындайтын программа құру қажет. Ол іс-әрекеттерге мыналар жатады: дерекқордағы бірін-бірі қайталайтын фактілердің бәрін өшіру және ол туралы хабарды экранға шығару. Сыртқы дерекқор нәтижесін сыртқы файлға жазу.

Жұмыстың орындалуы.

```
DOMAINS                          /* деректер типтерін анықтау бөлігі */
    list = string*.
DATABASE                         /* сыртқы дерекқор бөлігі */
    country (string).
PREDICATES                       /* предикаттар бөлігі */
    import.
    export.
    nondeterm del (string).
    add (list).
CLAUSES                          /* фактілері мен ережелер бөлігі */
    import: –
    consult («Lab6In.txt»),      /* файлды оқиды, ДҚ соңына қосады */
    write («Database import from file!»), nl.
    export: –
    save («Lab6Out.txt»),        /* файлды сақтайды */
    write («Database export in file!»), nl.
    del (H): – retract (country (H)).    /* ДҚ бірінші фактіні өшіреді */

/* фактіні тізімге қосады */
    add ([H | T]): –
    country (H), !,
    write («Double fact is delete: «, H), nl,
    add (T).
/* жаңа фактіні ДҚ соңына қосады */
    add ([H | T]): –
```

```
    assertz (country (H)), !, add (T).
    add ([ ]).
GOAL                    /* Мақсат бөлігі */
    import,
    findall (H, del (H), T),
    add (T),
    export.
```

Мына «Lab6In.txt» атаудағы файлда келесі ақпарат бар:

```
country («Russia»)
country («USA»)
country («France»)
country («Breat Britan»)
country («Germany»)
country («USA»)
country («France»)
country («France»)
country («Germany»)
```

Нәтижелер. Дерекқордағы бірін-бірі қайталайтын фактілерді өшіру мысалы нәтижесінің экрандық түрі. (1.7.11-суретті қара). *Дерекқорда қалған жазбалар «Lab6_Exp.txt» файлында сақталады.*

1.7.11-сурет. Дерекқордағы фактілерді өшіру

Сыртқы «Lab6_Exp.txt» файлында сақталған Дерекқорда қалған жазбалар мынандай болады:

```
country («Russia»)
country («USA»)
country («France»)
country («Breat Britan»)
country («Germany»).
```

6-мысал. Қатарлар және файлдар. Пролог тілінде келесі іс-әрекетті орындайтын программа жазу қажет. Мәтінге енгізілген сөзді іздейді. Іздеу бірінші дәл келген әріптердің ең көп санына байланысты болады. Мәтін файл бойынша беріледі. Нәтижені жаңа файлда сақтау қажет.

Жұмыстың орындалуы.

```
DOMAINS                          /* деректер типтерін анықтау бөлігі */
    Str = string.
    Ch = char.
    Word = string.
    file = f1; f2.
PREDICATES                       /* предикаттар бөлігі */
    find (Word).
    search (string, string, integer).
CLAUSES                          /* фактілері мен ережелер бөлігі */
    find (Word): –
    openread (f1, «lab7in.txt»),     /* мәтінді файлдан ашып оқиды */
    openwrite (f2, «lab7out.txt»),       /* мәтінді файлға жазады */
    writedevice (f2),
    file_str («lab7in.txt», Str),
    str_len (Word, Len),
    search (Str, Word, Len),
    closefile (f1),
    closefile (f2).
    search (Str, Word, Len): –
    searchstring (Str, Word, Pos),
    writef («The word [%s] find on % position», Word, Pos);
    Len2 = Len-1,
    substring (Word, 0, Len2, Word2),
    search (Str, Word2, Len2).
GOAL                                     /* Мақсат бөлігі */
    write («*** Lab 7. String & File ***»), nl, nl,
    write («Enter the word: «),
    readln (Word),
    find (Word).
```

Пролог тіліндегі тапсырмалар

1. Пролог тілінде мына сұрақтарға жауап бере алатын: студенттің аты, туған жылы, тапсырған пәні, бағасы туралы мәліметі бар про-

грамма құру. Мысалы, предикатты study (name, subject, numb, data) деп құруға болады.

2. Пролог тілінде ер жынысты адамдар арасындағы туысқандық қатынастарды сипаттайтын Дерекқор құру. Тапсырмалар нұсқалары: кітапхана (кітап, автор, бағасы, шыққан_жылы); тұратын_жері (фамилия, көше, үй, пәтер); студент (фамилия, пән, баға, оқу_жылы); автомобиль (түрі, түсі, құны, параметрі); компьютер (түрі, фирма, процессор, құны); видео (аты, актер, жанр, жылы, прокат); аудио (әнші, диск, тираж, жылы); азық-түлік (атаулары, өндіруші, сақталу мерзімі, бағасы); дүкен (товар, елі, көлемі, бағасы);

3. Пролог тілінде «Крестер-нөлдер» ойынын іске асыратын программа құру. Матрица өлшемі 3x3.

4. Пролог тілінде «Теңіздегі соғыс» ойынын іске асыратын программа құру.

5. Пролог тілінде «Тәтті кулинар» деп аталатын сарапшы-аспаздың жұмысын іске асыратын программа құру.

6. Пролог тілінде 1994 ж. шыққан «Форд». Маркалы автомобильдер жайындағы мәліметтерді сақтайтын Дерекқор құру.

7. Пролог тілінде автомобильді жөндеу жайында есептеуді жүзеге асыратын программаны жазыңыз.

8. Пролог тілінде Пәтерді жөндеу (бөлмелермен), бөлмелер санын, өлшемдерін ойдан алып, тұсқағаз жапсыруға кететін шығындарды есептеуді жүзеге асыратын программаны жазыңыз.

9. Пролог тілінде квадраттық теңдеу түбірін есептеуді жүзеге асыратын программаны жазыңыз.

10. Пролог тілінде үшбұрыштың ауданын табатын программаны жазыңыз. Оның үш қабырғасы да белгілі деп есептейік (үш қабырғаның өлшемдерін ойдан енгізіңіз). Герон формуласын қолданған жөн.

11. 6-7 студентке арналған және бірнеше пәнді қамтитын тұжырым жазыңыз. Мұнда фамилия, пән аттары, мерзімі бірнеше рет қайталануы мүмкін.

12. Программаны жұмысқа қосып, әртүрлі сыртқы мақсатта сұранымдар беріп көріңіз: айнымалыларсыз; бір немесе бірнеше айнымалымен; жасырын айнымалылармен.

13. Программаға бірнеше ереже енгізіңіз, мысалы: біледі (X): – оқиды (математика, _, 5). Программалаушы (X): – оқиды (X, жүйелік программалау, Z, Y), Z > 3, Y >= 5.

14. Мақсаттың ережемен қалай салыстыратынын түсіндіріңіз. Программаға ішкі мақсатты қосып, өзгеріс енгізіңіз.

Бақылау сұрақтары

1. Пролог тіліндегі факт, ереже, айнымалы деген не?
2. Пролог тіліндегі терм және құрылым деген не?
3. Прологта жариялау операторларын шығаруды кай предикат көрсетеді?
4. Прологта салыстыруды орындайтын ішкі предикаттарды атаңыз.
5. Пролог тіліндегі функтор деген не?
6. Пролог тіліндегі программаларды өңдейтін қандай әдістер бар?
7. Прологтағы тізімнің басын және құйрығын анықтайтын бөлу әдісі.
8. Прологтағы сандарға арифметикалық операциялар ережелерін ата.
9. Пролог тіліндегі тізімдерді қосатын қандай операторлар бар?
10. Пролог тілінде кесу қалай пайдаланылады?
11. Пролог тіліндегі тізімдерді қосатын қандай операторлар бар?
12. Пролог тілінің теориялық негізі не?
13. Процедуралық және мағлұмдамалық стильдердің негізгі ерекшелігі қандай?
14. Пролог тіліндегі термдерді енгізу-шығару қалай іске асады?
15. Пролог тіліндегі домендер типтері қандай?
16. Пролог тіліндегі ережелер мен фактілер айырмашылығы неде?
17. Турбо-Пролог программасы құрылымы. Барлық бөліктер қызметтері.
18. Турбо-Прологтағы салыстыру принципі.
19. Пролог тіліндегі рекурсияның қайталаудан айырмашылығы.
20. Пролог тіліндегі тізімге жату предикаты.

2-бөлім. ФУНКЦИОНАЛДЫҚ ПРОГРАММАЛАУ

Қазіргі таңда программалық қамтамасыз ету құрамы күн өткен сайын күрделене түсуде. Құрылымы жақсы сипатталған программалық қамсыздандыруда бірнеше рет пайдаланылатын модульдерді қолдану болашақта программалауға кететін шығындарды азайтады. Әдеттегі программалау тілдерінде модульдік құрылымды программалар құруға тұжырымдамалық шектеулер бар. Функционалды тілдер бұл шектеулерді алып тастауға мүмкіндік береді. Программаның модульдік қасиетін арттыру үшін функционалдық тілдерде екі түрлі ерекшелік қарастырылады. Ол ерекшелікке жоғары дәрежелі функция және «жалқау есептеулер» деп аталатын есептеу түрін пайдалануды жатқызамыз. Жалпы, функционалдық атаудың өзінде программалаудың осы түрінің негізі жатыр. Өйткені программаның өзі функциядан тұрады. Программа өзіне қажетті бастапқы деректерді функцияның аргументі есебінде қабылдап алып, нәтижені функция нәтижесі есебінде береді. Әдетте, негізгі функция басқа функциялар терминдері аумағында анықталады, ал олар одан ары қарай тағы да басқа функция аумағында анықталып, осылайша жалғаса береді. Ең соңындағы функция ең төменгі деңгейдегі тіл бірлігі болады.

2.1. Функционалдық программалау тарихы

Функционалдық программалау негізін 1934 жылы А.Чёрч (Alonso Church) атамыз салды. Ол кісі ең алғаш рет лямбда-есептеулерін жиындар теориясын зерттеуге пайдаланды. Бұл есептеулердің негізін қарапайым функциялар теориясы деген атпен 1924 жылы М.Шейнфинкель (Moses Schonfinkel) зерттеген болатын. Ал А.Чёрч салып кеткен жол математика ғылымындағы іргелі зерттеулерге жатады, оның теориясы қазір әдебиетте А.Чёрчтің лямбда-есептеулері деп аталады. Алгоритм ұғымының бір нұсқасын қалыптастыратын рекурсия түсінігі лямбда-есептеулерде өзінің тамаша дәлелін тапты. Қазіргі кезде лямбда-есептеулер есептеу үдерісін сипаттауға арналған Тьюринг машинасы немесе Марков алгоритмі сияқты кеңінен тараған математикалық механизмдер бар.

Сонымен қатар басқа математиктер де лямбда-есептеулерге ұқсас әртүрлі құралдар мен формализмдерді ойлап тапты. 1940 жылы Хас-

келл Карри (Haskell Curry) айнымалысыз функциялар теориясын құрды. Қазір ол «комбинациялық логика» деген атпен белгілі. Бұл теория лямбда-есептеулердің жалғасы болып табылады.

Ғылымның осы жолын ары қарай Д.Тернер (David Turner) жалғастырып, 1960 жылы функционалдық программалаудың үлкен бір бағытына дара жол салды. Ол кісі комбинаторларды функционалдық программалау тілдері трансляторларының төменгі деңгейлі кодасы есебінде пайдалануды ұсынды. Жоғарыдағы аталған лямбда және комбинаторлық есептеулер дискреттік математиканың бір ғылыми бағыты болып саналады.

1960 жылы осы теориялар негізінде Джон Маккарти (John McCarthy) Лисп (Lisp – List Processing) деп аталатын программалау тілін құрды. Лисп тілі көптеген жылдар бойы функционалдық стильде программалаудың тілі есебінде жүріп отырды. Бұл тілде қолданылатын көптеген идеялар функционалдық программалаудың негіздерін құрды деуге болады. Мысалы, атап айтсақ, функцияның өзін есептеу үдерісін ұйымдастырудың негізгі механизмі ретінде қарау, атом ұғымын енгізу, деректер мен программаның негізгі типтері есебінде тізімдерді қабылдау сияқты көптеген тәсілдер осы стильдің классикалық тірегі болып калыптасты. Лисп тілінен кейінгі функционалдық тілдің бірі ретінде APL тілін атауға болады. Бұл тіл көптеген математикалық идеяларды іске асыруға көмегін тигізді және онда жиындармен жұмыс істейтін өте куатты механизм бар.

1970 жылдардың аяғында Дж.Бэкус (John Bekus) жоғарғы дәрежелі функциялармен және комбинаторлық логика нотациясына жақын синтаксиспен жұмыс істеуге мүмкіндік беретін аппликативті жүйелерді құру жөнінде ой желісін ұсынды. Бұл кісі кезінде сол жылдардағы программалаушы мамандарға кеңінен таныс Фортран тілін құрушы топтың жетекшісі болған. APL тіліндегі идеяларды одан ары қарай дамыта отырып, ол FP жүйелерінің негізін салды. Бұндай дамудың негізгі көздеген мақсаттарының бірі – фон Нейман архитектурасына негізделген тілдердің есептеу кезіндегі тиімсіз жақтарынан арылу болатын.

Функционалдық программалау зерттеулерінің тағы бір бағыты есебінде теоремаларды автоматты түрде дәлелдеу мәселесі болды. 1970 жылдар аяғы мен 1980 жылдардың басында Эдинбург университетінің Роберт Милнер (Robin Milner) бастаған ғалымдары ML (MetaLanguage – мета тіл) тілінің семантикасын анықтады. Бұл тіл жоғарғы дәрежелі функциялармен жұмыс істеумен қатар, типтерді автоматты түрде өзі

шығарып тұратын қасиетке ие. Яғни өрнек типін оны сипаттамай-ақ алып тұрады. Зерттеудің бұл бағыты типтеу үлгілері деген атқа ие болды.

Осы тілге ұқсас тілдің біріне Miranda тілі де жатады. Оны 1985-1986 жылдарда Дэвид Тернер құрды. Бұл *жалқау* функционалдық тілдердің ең алғашқы қарлығашы еді. Бұндай тілдер есептеу өрнектерін оларға қажеттілік туғанша тоқтатып тұру қасиетіне ие. Функционалдық тілдерге осындай жол табу – ондай тілдердің семантикасын қайтадан қарап, олардағы деректердің шексіз құрылымдарын өңдейтін механизмдерді жасауға мүмкіндік берді.

1980 жылдардың ортасында Ericsson компаниясының зертханаларында қатар есептеу үдерістерін басқаруға арналған тіл ойлап табылды. Ол өнеркәсіптерде, соның ішінде телекоммуникация саласында кеңінен қолданылуға ие болған Erlang тілі болды.

Осылайша функционалдық программалау зерттеулерімен айналысатын ғалымдардың әр топтары өз бетінше, өздеріне ыңғайлы тілдерді ойлап табумен айналысып кетті. Бұл ғылымның осы саласының одан ары қарай іргелі зерттеулеріне кедергі болумен бірге, әртүрлі тілдердің трансляторларын үнемі жөндеп, бір-біріне сәйкестендіру мәселесіне ұласып кетті. Осы жағдайды түзету үшін, функционалдық программалаудың алдыңғы қатарлы зерттеушілері осы тілдердің ең жақсы қасиеттерін бір тілге біріктіріп, жаңа әмбебап тілдің тууына себепкер болды. Ғалымдар тілдің атын осы функционалдық программалау аталарының бірі Карри Хаскелл атымен атауды жөн деп санады. Тілдің алғашқы нұсқасы 1990 жылдардың басында құрылды. Қазіргі кезде Haskell-98 стандарты пайдаланылады.

Ғылыми бастауын Ресейде алған тағы бір функционалдық программалау тілдерін атап өтсек. Ол Рефал (РЕкурсивных Функций АЛгоритмический язык) тілі. Бұл тілдің ең алғашқы нұсқасын 1960 жылдардың басында Валентин Турчин құрған. Ол алғашында басқа тілдердің семантикасын сипаттауға арналған мета тіл есебінде құрылған болатын. Рефал тілі – типсіз тіл. Тілдегі деректер типтері екі бағыттағы тізім түрінде көрсетіледі. Тізімнің мұндай түрі басқа тізімдерден ерекше түрде сипатталады. 1980 жылдардың ортасында В.Турчин қолдануға жарайтын Рефал-5 нұсқасын құрды. Қазіргі кезде Рефал-6 және Рефал Плюс деп аталатын нұсқаулар қолданыста бар. Бұл тіл оған арнайы суперкомпилятор құрылған функционалдық программалау тілдерінің алғашқысы деуге болады. Қазіргі кезде Рефал тілі xml-құжаттарын өңдеуге арналған тіл есебінде пайдаланылатын құрал ретінде қолданылып жүр. Өйткені осы тілдегі деректердің жалғыз және негізгі

типі нысандық өрнектер xml-құжаттарын пайдалануға ғажап түрде сәйкес келіп отыр.

2.2. Функционалдық программалаудың негіздері

Функционалдық программалаудың математикалық негіздерін зерттеуден бұрын ондағы функциялар мен деректердің типтерімен танысып өтейік. Бұл атаулардың функциялық түрлеріне: функционалдық тип, каррирлеу, аппликация, өрнекті жалпылау, жазба қысқартуы, лямбда-өрнектер жатады. Ал деректердің құрылымына: тізімдер, тізімдік құрылымдар, S-өрнектер жатады [14].

Функционалдық түрлері. *Функция типтері.* **A** және **B** типтеріндегі деректер бар делік. Бұл деректер сандар, жолдар, тізімдер, функциялар болуы мүмкін. Енді бір айнымалы **f** функциясын қарастырайық. Оның айнымалысының анықталу аумағы **A** типіне, өз мәндерінің анықталу аумағы **B** типіне жатады делік. Сонда функция типі мына өрнекпен **A→B** анықталса, онда оны ***функционалдық тип*** деп атаймыз. Егер **REAL** – нақты сандар, **POSITIV** – теріс емес нақты сандар, **INTEGER** – бүтін сандар типтерін білдірсе, онда мысалға ***Sin(x)*** функциясының типі **REAL→REAL** типінде, ал ***ln(x)*** функциясының типі **POSITIV→REAL** типінде болады. Жалпы жағдайда типтер белгілеуде «→» таңбасы кездессе, онда ол ***функционалдық типте*** болады. *Каррирлеу (каррирование).* Бұл тип атауын есептеу үдерісінің сипаттауын комбинаторлық логикаға негіздеушілердің бірі Хаскелл Карридің атымен аталған. Функционалдық тип ұғымын пайдалана отырып, көп айнымалы функцияны бір айнымалы функция есебінде қарастыру мүмкіндігі бар. Мысалы, қосу $x + y$ функциясын алайық. Оның префикстік жазбасы мынандай болады: ***add(x, y)***. Ал функционалдық типі келесі түрде **REALxREAL→REAL** өрнектеледі, мұндағы **x** белгісі декарттық көбейтуді білдіреді. Сонымен, егер функциясының типі $(A_1 x A_2 x A_n) \to B$ сияқты болса, онда оны мынандай $A_1 \to (A_2 \to \ldots (A_n \to B) \ldots)$. екі түрлі типтегі элементтер есебінде қарастыруға болады. Функцияны осылайша өрнектеу оны ***каррирлеу*** деп аталады. Жоғарыда аталған ***add(x, y)*** функциясының қандай типте болатынын оның жазба түрі білдіреді. Егер жазба түрі ***add(x, y)*** сияқты болса, онда функция типі **REALxREAL→REAL** түрінде болады да, ал жазба түрі ***add x y*** сияқты болса, онда функция типі **REAL→(REAL→REAL)** түрінде болады. *Аппликация және Өрнекті жалпылау.* Әдетте, ***f(a)*** деген өрнекті ***a*** аргументіне ***f*** функ-

циясын қолдану деп түсінеміз. Аргумент орнында, оны операнд деп атайды, кез келген өрнек, ал оператор орнында функция аты болады. Функционалдық программалауда ***f(a)*** функциясын ***f*** және ***a*** аргументтеріне аппликация (қолданба) операциясын қолдану нәтижесі деп қарайды. Ол былайша өрнектеледі: **f(a)↔apply(f, a)** де, осы операцияны ***аппликация*** деп атайды. Ал өрнекті жалпылау түсінігін былайша таркатамыз: жоғарыда аталған **apply** операциясы өзіндегі екі аргументін де есептеп шығарады және оның типі **(A→A)→(A→A)** түрінде болады. Осындай өрнектер, яғни барлық функциялар бір айнымалының функциясы есебінде қарастырылатын өрнектер, және функцияның аргументке қолданылуы **apply** операциясы іске асырылғандағы жағдайы ***оператор-операнд формасындағы өрнек*** деп аталады. *Жазба қысқартылуы.* Функционалдық программалауда **apply** операциясы анық түрде көрсетілмейді де, оның орнына мына жазба түрлері қолданылады:

((x)(y)) немесе ***(x y)*** және егер жақшаларды алып тастасақ ***xy***. Жақшалардын санын азайту мақсатында мынандай келісім болған:

a1 a2 ...an ↔ (...((a1 a2) a3)...an,

мұндағы жақшалар келісім бойынша солдан оңға қарай орналасады. Мысалы, мына өрнек бар делік: apply ***(apply (add, 3), 5).*** Мұндағы ***add*** таңбасын каррирлеу функциясы деп қарасақ, онда бұл өрнек екі нақты 3 және 5 сандарын қосу операциясын білдіреді. Бұл өрнектегі ең ішкі мәні ***apply(add, 3)*** өрнегінде болады, яғни ***(add, 3)*** бір аргументтік функция, оның мәні константа болады, оған ең сыртқы ***apply*** қосылып, ең соңғы мән 8-ге тең болады. Енді осы түсіндіру бойынша бұл өрнек былайша жазылады:

((add (3)) 5) ↔ add (3)(5) ↔ add 3 5.

Егер ***add*** таңбасы **REALxREAL→REAL** типіндегі функция есебінде қарастырылса, онда мынандай қысқартылған жазба түрі қолданылады: ***add(3, 5).*** Қорыта айтсақ, ***Жазба қысқартуы*** мынаны білдіреді: өрнектегі барлық функциялар бір айнымалының функциясы есебінде қарастырылса және функцияның аргументке қолданылуы ***apply*** операциясы көмегімен орындалса, онда ол өрнекті ***оператор-операнд*** қалыбындағы өрнек деп атаймыз. Мысалы, әдеттегі мынандай жазба бар делік: ***f(a) = sin(a + tg(b)) + tg(sin(a + b)).***

Онда оның ***оператор-операнд*** қалыбындағы өрнегі келесі түрде болады:

$$f\ a\ b = add(sin(add\ a\ tg(b)))(tg(sin(add\ a\ b))).$$

Аргументтері немесе мәндері функция болатын функцияларды ***жоғарғы тәртіптегі функциялар*** деп атаймыз.

Лямбда-өрнектер. Әдеттегі математикалық мәтіндерде функцияны анықтау былайша өрнектеледі: $f(x) = a + bx$ де, оның үш құрамдас бөлшегі болады: ол f – функцияны белгілейтін таңба, оны функция сәйкестендіргіші деп атайды, екіншісі – (x) – аргументтер (айнымалылар) тізімі және оң жақта орналасқан өрнек, оны есептеу арқылы белгілі бір мәнді таба аламыз. А.Чёрч (Alonso Church) анықтамасы бойынша функцияны келесі түрде жазамыз:

$$f = \lambda x.a + bx$$

мұндағы теңдеудің сол жағында f – сәйкестендіргіш, оң жағындағы өрнек ***лямбда-өрнек*** деп аталады. Лямбда-өрнек құрамында «λ» таңбасы бар, одан кейінгі орналасқан x айнымалысын байланған айнымалы деп, ал «.» таңбасынан кейінгі өрнекті лямбда-өрнектің денесі деп атайды.

Есептеулер орындалғанда лямбда-өрнектер үш түрлі жағдайларда қолданылады:

Лямбда-өрнектер – оператор есебінде. Егер лямбда-өрнектер белгілі бір операндыға оператор есебінде қолданылса, онда мынандай амалдар орындалады: байланған айнымалы лямбда-өрнек денесіне еніп, содан соң дене мәні есептеледі. Бұл ереже әдеттегі функция мәнін «мән бойынша шақыру» сызбанұсқасы бойынша өтеді. Мысалы, мына өрнек $(\lambda x.\ x\char`^2 + 3)5$ бар делік. Аталған ережені пайдаланып, мынаны аламыз: $5\char`^2 + 3 = 28$. Осындай амалдар әдеттегідей $f(5)$ функциясын есептегенде де орындалады. Яғни функция мәні мына формуламен $f = \lambda x.\ x\char`^2 + 3$ немесе мына өрнекпен беріледі:

$$f\ x = x\char`^2 + 3$$

Лямбда-өрнектер – операнд есебінде. Бұл жағдайда лямбда-өрнектің мәні есептеледі (ол мәннің өзі лямбда-өрнек болуы мүмкін) және ол байланған айнымалының орнына оператор орнында тұрған

лямбда-өрнек денесіне кіреді. Мысалы, мына өрнектер осы есептеу түрін көрсете алады:

$$(\lambda f, f5)\ лx.\ x\text{^}2 + 3 = (\lambda x.\ x\text{^}2 + 3)5 = 28.$$

Лямбда-өрнектер –басқа лямбда-өрнек денесі есебінде. Қолданудың мұндай түрі көп айнымалы функцняны үлгілеуге мүмкіндік береді. Мысалы, мынандай функция $g = x\text{^}2 + y\text{^}2$ бар делік. Бұл функция типі келесі **REAL→(REAL→REAL)** тнпінде болады, яғни мысалдағы функцияны мынандай лямбда-өрнек түрінде бере аламыз: $g = \lambda x.\ (\lambda y.\ x\text{^}2 + + y\text{^}2)$. Аталған ережелерді қолданамыз:

$$((g3)4) = ((\lambda x.\ (\lambda y.\ x\text{^}2 + y\text{^}2))3)4) = (\lambda y.\ 3\text{^}2 + y\text{^}2)4 = 3\text{^}2 + 4\text{^}2 = 25.$$

Сонымен, қорыта айтсақ, аппликация, каррирлеу, лямбда-өрнектер ұғымдары функция түрлерін анықтауда маңызды рөл атқарады. Аппликация конструкциясын енгізу өзінің екі аргументін (оператор және операнд) бірдей есептейтін функцияларды есепке алуды жеңілдетеді. Яғни бұл жағдайда функция аргументі не оның мәні есебінде басқа функция бола алады. Көптеген аргументтері бар функцияны бір аргументті жоғарғы тәртіпті функция есебінде көрсете аламыз. Функцияның осылайша анықталуы оны каррирлеу деп аталады. А.Чёрч анықтаған лямбда-өрнектер аппликация және каррирлеу түсінігімен қосылып, жоғарғы тәртіпті функцияны анықтаудың және онымен іс-әрекет жасаудың негізін қалыптастырады.

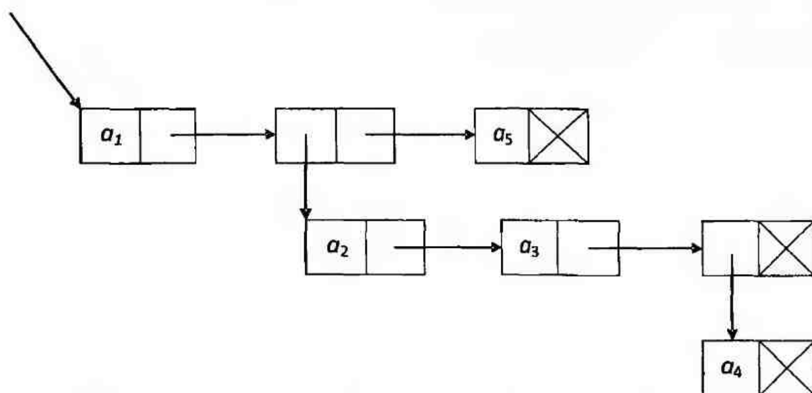
Деректер құрылымдары. Программалаудың функционалдық тілдерінде әдетте таңбалар және сандармен жұмыс істейді. Осы тілдердің негізін салған Джон Маккартиден келе жатқан дәстүр бойынша таңбалар мен сандарды атомдар деп атайды. Ол бұл таңбалардың бөлшектерге бөлінбейтін бүтіндік қасиетін білдіреді. Осы Маккарти атамыз «тізім» және оның кеңейтуі «тізімдік құрылым» деген ұғымдарды енгізген. ***Тізімдер.*** Бұл құрылымдар негізінде ***x:y*** және жұптарының «жұп құру» операциясы бар. Операцияның осылайша жазылуы «инфикс қалыбында» деп аталады. Ал осы операцияның «префикстік» түрі келесі түрде: ***prefix x y*** жазылады. Бұл ***Prefix*** операцияның нәтижесі екі базистік ***head*** (*басы*) және ***tail*** (*құйрығы*) операциялары арқылы анықталады. Әдебиеттерде қысқаша бұл екі операция ***h*** және ***t*** деп белгіленеді. Мысалы, егер $z = x{:}y$ болса, онда оны былайша ***head*** $z = x$ ***tail*** $z = y$ өрнектеуге болады. Бұл операциялардың

тағы селекторлар деген атауы да бар. Өйткені олар бүтіннен бөліктерді алып шығады. Жоғарыдағы айтылған операцияларды тұжырым түрінде жазған ыңғайлы болады. Яғни:

h(x:y) = x
t(x:y) = y
(h(x:y) : (t(x:y)) = x:y

S-өрнектер. Жалпы жағдайда функционалдық программалау ортасында атомдардан құралатын жиындардың элементтерін *S-өрнектер* деп атайды. Әдетте, программалауда бұндай өрнектердің кіші жиындары тізімдер ***(List)*** мен тізімдік құрылымдары ***(ListStructure(A))*** қолданылады. Мысалы, мынандай тізімдік құрылымды қарастырайық: $[a_1, [a_2, a_3, [a_4]], a_5]$. Бұл тізімдік құрылым *S-өрнектерді* құрады.

Тізімдік құрылымдар. Тізімдік элементтердің бәрінің типтері біреу болады. Мысалы, *A*-типі. Ол былайша белгіленеді: ***ListStructure(A).*** Бұндай тізімдік құрылымдар үшін *«ену деңгейі»* деп аталатын ұғым бар. Ол ең көп ***«[»*** ашылу жақшасының санына тең болады. Мәселен, жоғарыдағы тізім үшін ол сан элементі a_4 үшін 3-ке тең, ал a_5 элементі үшін 1-ге тең. Функционалдық программалау тәжірибесінде атомдарға машина зердесінде белгілі бір ұяшықтар бөлінеді. Мысалы, *z* нысаны екі жұп ***x*** және ***y*** ұяшығының адресін білдіреді [14]. Жоғарыдағы келтірілген мысалдың компьютер зердесіндегі орналасу сызбанұсқасын келесі сурет анықтап бере алады.



2.2.1-сурет. Мысалдағы $[a_1, [a_2, a_3, [a_4]], a_5]$ тізімдік құрылымның графикалық бейнеленуі

Мұндағы ұяшықтар жұбындағы ену «↓» және шығу «→» бағдаршалары нысанның *«басын»* және *«құйрығын»* білдіреді. Ал «Х» белгісі бос тізімді көрсетеді. Осы аталған бағдаршаларды әдебиетте *а-өріс («басы»)* және *d-өріс («құйрығы»)* деп атайды. Сызбанұсқада көрсетілгендей, жұпты компьютер зердесінде сақтау үшін бағдаршаларға екі ұяшық керек. Бұл есептеулерге көп компьютер зердесін қажет етеді. Сондықтан *а-өріс* және *d-өрістеріне* тізімнің мазмұнының өзі жазылады да, оған қатынауда ***h*** және ***t*** операциялары қолданылады. Бұндай әдіс функционалдық программалауда компьютер зердесін үнемдеуге өз әсерін тигізеді.

Сонымен, қорыта айтсақ, сандық емес ақпаратпен жұмыс істеу үшін тізімдер, тізімдік құрылымдар, S-өрнектер деп аталатын құрылымдар қолданылады. Бұндай деректермен жұмыс істегенде, префикстік түрдегі ***prefix*** конструкторы және инфикстік түрдегі «:» конструкторы қолданылады. Сонымен бірге жұптың бірінші және соңғы элементін бөліп қарауға арналған екі селектор (***head*** қысқаша ***h***) және (***tail*** қысқаша ***t***) пайдаланылады. Тізімнің басы болып ***A*** типіндегі элемент, ал құйрығы болып ***List(A)*** (оның ішінде бос тізім) типіндегі элемент табылады.

Функционалдық программмалаудағы нотациялар. Программалаудың аталған стилінде оның көптеген тілдеріне арналған жалпы нотациялар бар. Нотация ұғымы осы программалау стиліне жататын программалау тілдерінің бәріне ортақ белгілі бір белгілеу ұғымдарын білдіреді. Оны осы тілдердің псевдокоды[1] деп те атауға болар еді. Осындай нотацияларға: көрсетім үлгілері, клоздар, лямбда-өрнектер, жергілікті айнымалылар, қорғау нотацияларын жатқызамыз. Енді осыларға қысқаша тоқталып өтелік.

Көрсетім үлгілері, клоздар. Әдеттегі программалауда алгоритм тармақталуы мына ***if-then-else*** конструкция көмегімен өтеді. Функционалдық программалау тілдерінде алгоритм тармақталуын ұйымдастырудың басқа технологиясы қолданылады. Ол конструкцияны клоз деп атайды. ***Клоз*** (ағылшын. ***clause***) дегеніміз көрсетім үлгілерімен анықталған белгілі бір функцияның есептеу нәтижесінің бір нұсқасының жазбасы болып табылады. Клоздың Бэкус-Наур нотациясы бойынша жазбасы келесі түрде болады:

<clause > : : = <faunction_name > <patterns > = <expression>

---

[1] Псевдокод – программалау тілдерінің көптеген синтаксистік мағлұматтарын алып тастап, оның негізгі сөздерін пайдалана отырып, алгоритм сипатын беретін тіл.

мұндағы ***faunction_name*** – функция атауы; ***expression*** – тіл синтаксисі тұрғысынан дұрыс құрылған белгілі бір өрнек; ***patterns*** – салыстырылуға арналған көрсетім үлгілернің тізбектері. ***Көрсетім үлгілері*** (ағылшын. ***pattern***) деп деректерді салыстыруға пайдаланылатын оларды құрастыру операциясының өрнектерін айтамыз. *Көрсетім үлгілеріне* қойылатын талап біреу: деректерді көрсетім үлгілерімен салыстырғанда, көрсетімге енетін айнымалыларды *таңбалау* тек бір-ақ рет болады. Мысалы, егер мына ***(1 + x)*** өрнекті көрсетім есебінде 5 санымен салыстыру үшін, ***x*** айнымалысы үшін таңбалау бір ғана ***(x = 4)*** мәнде болады. Ал ***(x + y)*** өрнегін көрсетім есебінде пайдалануға болмайды, өйткені 5 константасымен салыстыруда ***x*** және ***y*** айнымалыларын таңбалауды әртүрлі жолдармен іске асыруға болады. Бұл мысал бойынша көрсетім үлгілері есебінде мыналар жүре алады: 5 – сандық константа; ***x*** – айнымалы; ***[ ]*** – бос тізім; ***(x:xs)*** – тізім, ***x*** – басы бар және ***xs*** – құйрығы бар, бос емес тізім; ***x(y:[ ])*** – тізім, ***x*** және ***y*** екі элементі бар, тізім; ***[x, y]*** – осындай тізім, бірақ басқа қалыпта жазылған. *Лямбда-өрнектер.* Функционалдық программалауда лямбда-өрнектерді функцияны анықтауда пайдалануға болады. Мысалы математикада кеңінен таныс факториал функциясын қарастырайық. Оның математикалық түрі мына формуламен анықталады:

$$fact(n) = \prod_{i=1}^{n} i$$

Бұл формуланы лямбда-өрнек арқылы былайша жазамыз:

fact = λn. If n = 0 then 1 else n*fact(n-1).

Әрине, бұл жағдайда клоздық жазбадан бас тартамыз. Енді функцияны былайша анықтайық:

map f [ ] = [ ]
map f (x:xs) = (f x): map f xs

Бұл программалаудағы кеңінен таныс «бейнелеу» функциясы. Оның негізгі қызметі: элементтер тізімінің орнына әр элементке ***f*** функциясын қолданғанда пайда болатын нәтижелер тізімін қайтарады. Енді мынандай ***[1, 2, 3, 4]*** тізім бар делік. Оның элементтерін квадраттау үшін мына амалды орындаймыз:

map(λn.n*n) [1,2,3,4],

нәтижесінде мынандай ***[1, 4, 9, 16]*** тізім аламыз.

Жергілікті айнымалылар. Бізге мынандай ***h(f x)(f x)*** функцияны есептеу қажет болсын дейік. Егер тіл конструкциясында есептеулерді оңтайландырулар қарастырылмаса, онда «мәні бойынша шақыру» іске асқанда, ***f x*** функциясы екі есе есептеуге кетіп қалады. Осыдан құтылу үшін екі форманы қабылдайды: префикстік, негізгі ***let, in*** сөздері және постфикстік, негізгі ***where*** сөзі. Ол алғашқы өрнекті мына түрде жазуға мүмкіндік береді. Префикстік түрі: ***let v = f x in h v v;*** және постфикстік түрі: ***h v v where v = f x;***. Бұл формалардың негізінде жергілікті айнымалыны пайдалану жатыр. Біздің жағдайда бұндай айнымалы есебінде ***v*** жүреді. Яғни ***let*** (немесе ***where***) деген сөздерден кейін қанша қажет болса, сонша жергілікті айнымалыларды анықтауға болады.

Қорғау. Қорғау шарты мына негізгі ***when*** және ***otherwise*** сөздермен енгізілетін логикалық өрнек арқылы анықталады. Көрсетім үлгілерінің механизмі есептеу үдерісінің тармақталуын жақсы сипаттайды. Дегенмен ол кейбір жағдайда жеткіліксіз болып қалады. Мысалы, сандық тізімдерге белгілі бір операцияларды орындағанда, сан оң не теріс болса, әртүрлі амалдар орындау қажеттігі туады. Осы жағдайда клоздың оң жағында шартты өрнекті қолдануға болады. Осы механизмді ***қорғау*** деп атаймыз. Мысалы ***onpos*** функциясын (тек қана оң сандар) қарастырайық. Оны қорғауды былай жазуға болады:

onpos [ ] = [ ]
onpos (x:xs) = x : (onpos xs)when x > 0
onpos (x:xs) = onpos xs otherwise.

Сонымен, қорыта айтсақ, функционалдық программалау неѓіздерін оқып үйренуде дерексіз нотация түрлері қолданылады. Бұнда функцияны анықтауда клоздық жазба түрі пайдаланылады. Клоздық жазбада мына ереже бар: нақты параметрлі функцияға қатынау болғанда, бірінші тәртіптегі клоз ізделіп, оның параметрлері көрсетім үлгілерімен дәл келуі керек. Салыстыру кезінде қорғау шарты шын мәнінде болуы қажет.

2.3. Функционалдық программалау ерекшеліктері

Функционалдық программалауда меншіктеу операторы жоқ және айнымалылар бір рет мән қабылдаған соң өзгермейді. Функцияға қатынауда тек нәтиже ғана аламыз. Программалаушы адам басқару

ағынын сипаттау қиындығынан құтқарылады. Берілетін өрнектер кез келген уақытта есептеле алатындықтан, айнымалыларды олардың мәнімен алмастыруға болады және керісінше амалды да орындауға мүмкіндік бар. Программа сілтемелерінің мұндай анық, әрі «мөлдір» болуы функционалдық программаларға математикалық өңдеуді қолдану мүмкіншілігін береді [15].

Енді *құрылымдық программалау* деп аталатын программалау түрінен функционалдық программалау ерекшеліктерінің қандай екенін қарастырайық. Құрылымдық программалауда goto операторлары жоқ. Бұл тәсілде енулер мен шығулардың бірнешеуі бар. Сондықтан олар математикалық өңдеуге ыңғайлы. Сонымен қатар *құрылымдық программалауда* оны құруда *модульдік жол* қарастырылған. Ол жобаларды құрғанда, еңбек өнімділігін өсіруге мүмкіндік береді. Өйткені аз ғана мөлшерлі модульдерді кодалау оңай, әрі жеңіл. Екіншіден, жалпылық қасиеттегі модульдерді бірнеше қайтара пайдалануға болады. Үшіншіден, модульдегі *қателіктерді* басқа модульдерге тәуелсіз түрде тексеріп, жөндеуге болады. Қазіргі кезде программаларды модульдік түрде құру жетістікке жеткізу жолдарының бірі ретінде қарастырылуда. Мына тілдер құрамында: Modula-2 [Wir82] , Ada [OD80] және Standard ML [MTH90] модульдерді жақсартатын конструкциялар бар. Дегенмен модульдік программалар құрғанда мәселені шешу үшін, «үлкен есепті кіші есептерге бөлшектеу» деген сияқты принципті ұстану қажет. Содан соң осы бөлшекті біріктіру мәселесі туады. Бұл проблеманы өмірдегі мынандай мысалмен өрнектеп жеткізуге болады. Ағаштан орындык жасау үшін, алдымен оның: төрт аяғын, отырғышын деген сияқты бөліктерін жасаймыз. Енді оны біріктіру үшін жоғары сапалы жабыстырғыш (клей) болса жеткілікті, біз тез арада дайын орындықты аламыз. Егер аталған бөлшектердің кейбірі болмаса, біз орындықты бітеу ағаштан ойып жасаймыз. Ал бұл күрделі есепке жатады. Айтылған мысал модульдік программалаудың жақсы жағымен қоса, кемшіліктерін де көрсетеді. Енді біз өзіміз қарастырып отырған *функционалдық программалауға* қайта оралайық. Функционалдық программалауда модульдерді біріктіретін *екі түрлі* өте маңызды *элемент* бар екенін айтып өтуіміз қажет. Бұл элементтер функционалдық программалау қуатын арттырып қана қоймай, көлемі аз, қарапайым модульдерді құруға мүмкіндік береді [16].

Біріктірудің бірінші түріне: *«Функцияларды байлау, біріктіру»* деп аталатын тәсіл жатады. Оған қарапайым функцияларды одан гөрі күрделі түріне байлау жатады. Мысал қарастырайық. Тізімдерді өңдеу, олардың элементтерін қосу. Біз тізімдерді былайша анықтаймыз:

data [x] = [] | x : [x]

Бұл мынаны білдіреді: икстер тізімі не бос тізім [], не басқа *x*-тер тізімдердің конструкциясы. Берілген ***x:xs*** тізім болып табылады, оның бірінші элементі – ***x***, ал келесілері – ***xs*** тізімінің элементтері. Бұндағы ***x*** элементі кез келген типте болуы мүмкін, мысалы, ***x*** «бүтін сан» болса, онда берілген анықтама бойынша бүтін сандар тізімі не бос тізім, не бүтін сандардан құралатын басқа бүтін сандар тізімінен тұрады. Тізімдерді олардың элементтерін тік жақша ішінде жазу арқылы береміз. Мысалы, [1] мынаны білдіреді 1:[]. Ал [1, 2, 3] деген 1:2:3:[] білдіреді. Тізім элементтерінің қосындысын sum рекурсивті функциясы көмегімен алуға болады. Ол екі түрлі параметр үшін анықталады: бос тізім және конструктор үшін. Бос жиындар қосындысы нөлге тең болатындықтан, біз мынаны анықтаймыз:

sum [] = 0

Қосынды тізімнің бірінші элементін басқа элементтердің қосындысына қосу арқылы анықтаймыз:

sum num:list = num + sum list

Бұл жерде («0» және «+») қосындыны анықтайтын элементтерге жатады. Ол – қосындыны есептеу модулінде «бөлінген бөліктер» мен «рекурсивтік түр» бар деген сөз. Ол әдетте орау (reduce) деп аталады да, былайша өрнектеледі:

sum = reduce (+) 0

Орауды sum функциясын параметрлеу арқылы алуға болады:

reduce f x [] = x

reduce f x (a:l) = f a (reduce f x) l

Одан соң біз (reduce f x) тізбекшесін бөліп аламыз. Оны ол ***sum*** функциясын ауыстыра алатынын білу үшін орындаймыз. Осы үш аргументтік (reduce f x) типіндегі функция екі аргументке қолданылса, онда ол **l** параметрлі функция болып табылады. Жалпы жағдайда ***n*** параметрлі функция ***m (m < n)*** параметріне қолданылса, онда ол қалған ***n–m*** аргументтерінің функциясы болып табылады. Осылайша модульденген *sum* функциясын біз бірнеше рет пайдалана аламыз. Бұндағы *reduce* функциясы ең ғажап функцияларға жатады. Оны тізім элементтерін көбейтуге де пайдалана аламыз. Мысалы:

product = reduce (*) 1

Немесе тізімдегі айнымалылар қабылдайтын мәндер «буль» түріне жата ма, соны тексеруге болады:

anytrue = reduce (||) False

Немесе олардың барлығы «шын» мәнін қабылдай ма:

alltrue = reduce (&&) True

Осы *reduce f a* мәнін функция есебінде пайдаланудың бір жолына ойша ондағы «:» таңбасын ***f*** таңбасына ауыстыру және барлық «[]» таңбасын ***a*** таңбасына ауыстыру жатады. Мысалы, мына берілген тізім *1 : 2 : 3 : []: reduce (+) 0* мына түрге түрленеді де, *1 + 2 + 3 + 0 reduce (*) 1* ал өз кезегінде **1* 2 * 3 * 1** түрге түрленеді. Бұл түрленуден біз мынаны түсінеміз: *reduce (:)* **[]** функциясы тізімді қарапайым түрде қайтадан көшіріп жазады. Тізімді оның элементтерін алдыға орналастыру көмегімен басқа тізімге қосып, біз мынаны аламыз:

append a b = reduce (:) b a

Тізімдегі элементтерді екі еселеу үшін, мына функцияны қолданамыз:

doubleall = reduce doubleandcons []

doubleandcons num list = 2 * num : list

Өз кезегінде, *doubleandcons* функциясы модульдеуге жатады:

doubleandcons = fandcons double

double n = 2 * n

fandcons f el list = (f el) : list

содан соң

fandcons f = (:) . f

мұндағы *«.» – (f . g) h = f (g h)* теңдеуі бойынша анықталған функционалды композицияның стандарты операторы. Бұл *fandcons* функциясын кейбір параметрлерге қолданып көрелік:

fandcons f el list = (f el): list

одан ары қарай

doubleall = reduce ((:) . double) []

Келесі үлгілеуде біз мынаған жетеміз:

doubleall = map double

map f = reduce ((:). f) []

Аталған *map* функциясын кез келген *f* функциясын тізімнің барлық элементтеріне қолданамыз. Мысалы, матрица элементтері тізімдерден тұратын тізім есебінде берілсе, онда осы элементтер қосындысын алу үшін келесі функцияны қолдануға болады:

summatrix = sum . map sum

Мұндағы *summatrix* көрсетілген *(map sum)* функциясы барлық қатарларды қосу үшін пайдаланылады, ал ең шеткі **sum** функциясы нәтижелерді өзара қосып, бүтін матрицаның қосындысын алады. Тағы бір мысал ретінде «реттелген ағаш» деп аталатын деректер типін қарастырайық:

data Treeof x = Node x [Treeof x]

Бұл анықтама бойынша, ***x*** элементтер ағашы – ***x*** таңба болып табылатын түйін, ал ағаштар тармағының өзі де тізім болады да, оның өзі де ***x*** элементтер ағашы болып табылады. Оны түсіну үшін, *reduce* функциясына ұқсас *redtree* функциясын қарастырайық. Аталған *reduce* функциясы екі параметрді қабылдап алатын – оның бірі «:» таңбасын ауыстырады, келесісі «[]» таңбасын ауыстырады. Ағаштар *Node* функциясын пайдалану арқылы қалыптасатындықтан, *redtree* функциясы үш параметрді қабылдап алуы қажет. Ағаштар мен тізімдердің типтері әртүрлі болғандықтан, біз екі функцияны анықтауымыз қажет:

```
redtree f g a (Node x subtrees) = f x (redtree' f g a subtrees)
redtree' f g a (x:rest) = g (redtree f g a x) (redtree' f g a rest)
redtree' f g a [] = a
```

Аталған *redtree* функциясын басқа функциялармен байланыстыра отырып, көптеген функциялар түрлерін ала аламыз. Мысалы, ағаштарда орналасқан сандарды қосу арқылы барлық элементтер қосындысын алуға болады:

```
sumtree = redtree (+) (+) 0
```

Ағаштардағы барлық элементтер тізімін алуға болады:

```
labels = redtree (:) append []
```

Ең соңында *map* функциясына ұқсас ағаштың барлық элементтеріне қолдануға болатын ***f*** функциясын алуға болады:

```
maptree f = redtree (Node. f) (:) []
```

Жоғарыда келтірілген функцияларды алу мүмкіндіктерінің барлығы функционалдық тілдер көмегімен іске асады. Өйткені онда басқа программалау тілдерінде болмайтын қасиет бар. Ол қасиетке кез келген функцияны белгілі бір бөлшектерге бөліп қарауға да, біріктіріп қарауға да болатын мүмкіндік жатады. Мұндай жоғары деңгейдегі функциялар көптеген операцияларды программалауға мүмкіндік береді. Әр кезде деректердің жаңа типі анықталған сайын оны өңдейтін жоғары деңгейлі функциялар жазылады. Бұл деректер типімен жұмыс істеуге ыңғайлы және осыған қатысты білімдерді бір жерде жинақтап ұстау үшін қажет.

Біріктірудің екінші түріне: *«Программаларды байлау»* деп аталатын тәсіл жатады. Функционалдық программа деп ену деректері мен шығу нәтижесі бар программаны айтамыз. Егер ***F*** және ***G*** осындай программа болса, онда ***(G. F) – G (F input)*** функциясын есептейтін программаға жатады. Аталған ***F*** программасы өз нәтижесін есептейді, оның нәтижесі ***G*** программасына арналған ену дерегі болып табылады. Оны іске асыру үшін ***F*** программасындағы нәтижені уақытша

файлға орналастыру арқылы орындауға болады. Бірақ мұндай уақытша файлдар жадының көп көлемін алады, сондықтан оны шешу үшін функционалдық тілдерде мынандай амалдар қарастырылған. Аталған екі ***F*** және ***G*** программа бірге синхронды түрде орындалады. Көрсетілген ***F*** программасы аталған ***G*** программасы белгілі бір ену деректерін оқығанда іске қосылып, ол аяқталғанша орындалып тұрады. Одан соң ***F*** программасы тоқтап тұрады да, ***G*** программасы іске қосылады, ол кезекті деректер тобын оқиды. Егер ***G*** программасы ***F*** программасының барлық шығу деректерін оқып болмай аяқталса, онда ***F*** программасы үзіледі. Тіпті ***F*** программасы шексіз шығулары бар аяқталмайтын программа болса да, ол ***G*** программасы аяқталғанда тоқтайды. Ол цикл денесінен шығу шартын анықтайды. Ал оның өзі модульдегі аса қуатты құрал болып табылады. Бұл әдіс «жалқау есептеулер» деп аталады. Өйткені ***F*** программасының орындалуы қанша мүмкін болса, сонша сирек орындалады. Осы «жалқау есептеулер» әдісі функционалдық программалаудың ең қуатты құралына жатады [17].

Енді жасанды зерде саласынан бір мысал қарастырайық [18]. Жоғарыда атап өткендей, функционалдық программалаудың қуатты құралдарына екі түрлі байлау элементін жатқыздық. Олар: жоғары дәрежедегі функциялар мен «жалқау есептеулер». Жасанды зерде саласына қатысты бір алгоритм мысалын қарастырайық. Ол – «альфа-бета эвристика», ойыншы орнын бағалауға арналған алгоритм. Алгоритм ойынның қалай дамуын көріп отырады және тиімсіз нұсқаларды талдамай, мүмкін болатын жүру қадамдарын бақылау арқылы жұмыс істейді. Ойындағы орындар position типіндегі нысандармен сипатталған дейік. Бұл тип ойыннан ойынға өзгеріп отырады, бірақ біз ол жөнінде ешқандай жорамал жасамаймыз. Бірак осы бағытта қандай қадамдар болатынын білдіретін әдіс болуы қажет. Мынандай функция: *moves:: Position → [Position]* бар делік. Бұл функция ойын позициясын параметр есебінде қабылдап алып, бір қадамда іске асатын әрекеттер тізімін қайтаруы керек. Әрекет орны аркылы келесі жүру кезегі кімдікі екенін біліп отыруға болады. Оны «Крест-нөл» ойынында крестер мен нөлдер санын санау арқылы білуге болады. Шахмат ойынында мағлұмат *Position* типіне енгізілуі қажет. Ең алдымен ойын ағашын қалыптастыру қажет. Ондағы түйіндер – әрекеттер де, ал түйіннің балалық элементтері аталық түйіннен бір кадам жасау нәтижесінде іске асып отырады. Яғни түйін *p* әрекетінде болса, онда оның еншілес элементтері *(moves p)* әрекетінде болады. Ағаш шексіз болуы мүмкін. Ойын ағашы *moves* қайталап қолданудан қалыптасады. Ең түпкі

әрекеттен бастап, түп-тамыр тармақтарын құратын таңбаларды жасау үшін *moves* пайдаланылады. Одан соң осы moves одан ары қарай тармақталатын тармақтарды құруға пайдаланылады. Осылайша жалғаса береді. Бұл рекурсивті үлгі жоғары деңгейдегі мына функциямен өрнектеледі:

reptree f a = Node a (map (reptree f) (f a))

Бұл функцияны пайдалана отырып, басқа функцияны құруға болады: *gametree p = reptree moves p.* Ол белгілі бір әрекеттен бастап ойын ағашын құрады. Мұндағы *(reptree)* функциясы *iterate* (ол шексіз тізімдер құрады) функциясына ұқсас. Алгоритм ойын ағашын аталған әрекеттен бастап қарап шығады да, ойынның одан ары қарай даму бағытын бақылауға тырысады. Оны орындау үшін әрекет мәнін алдын ала тексермей тұрып-ақ бағалайтын бір белгі болуы қажет. Бұл «статикалық бағалау» алдыңғыларды қарау шекарасында болуы керек. Бұндай бағалау белгісі компьютер тарапынан болатын жауап өлшемі ретінде жүреді. (Ойында компьютер мен адам ойнайды деп шешілген). Нәтиже көп болған сайын компьютер әрекеті жақсара береді. Ең қарапайым жағдайда мұндай функция компьютер жеңгенде **+1** мәнін қайтарады да, керісінше жағдайда **–1** мәнін қайтарады, ал басқа жағдайларда **0** мәнін қайтарады. Мысалы, мына функция: *static: Position → Number* бар делік. Ойын ағашы *Treeof Position.* Ол *maptree static* функциясы көмегімен *Treeof Number* түрленеді. Аталған *maptree static* функциясы ағаштағы барлық позицияларды статикалық түрде есептейді (олардың саны шексіз болуы мүмкін). Осының ішіндегі «шын» мәнінде болатын әрекетті қалай табуға болады? Түпкі әрекет мәні қандай? Осы сұрақтарға жауап табу үшін түйін мәнін оның еншілес түйіндерінен іздеу қажет. Ол үшін әр ойыншының кезекті жүрісін ең жақсы жүріс деп есептейміз. Компьютер үшін ең үлкен мән ең жақсы болып табылатындықтан, ол ең үлкен мәнді еншілес түйінге баратын жолды таңдап алады. Ал оның қарсыласы ең аз мәндегі жолды таңдайды. Түйін мәндерін компьютер үшін *maximise* функциясы есептейді, керісінше жағдайда *minimise* функциясы есептейді. Яғни: *maximise (Node n sub) = maximum (map minimise sub) minimise (Node n sub) = minimum (map maximise sub).* Мұндағы *maximum* және *minimum* – тізімдегі максимум және минимум мәнін қайтаратын тізімдерден тұратын функция. Жоғарыда келтірілген статикалық бағалау ойыншы жеңгенде немесе ойын шекарасының маңында қолданылады. Аталған функциялардың ең соңғы анықтамалары:

maximise (Node n []) = n maximise (Node n sub) =
= maximum (map minimise sub) minimise (Node n []) =
= n minimise (Node n sub) = minimum (map maximise sub)

Келген позицияның мәнін қайтаратын функция түрін де жазуға болады: *evaluate = maximise. maptree static. gametree*; Бірақ бұл анықтаманы қолданудың өзіндік қиындықтары бар. Біріншіден, ол шексіз ағаштар үшін жұмыс істемейді. Аталған *maximise* функциясы рекурсивті түрде орындала береді де, ағаш соңы болғанша істейді. Егер соңы болмаса, бұл функция ешқандай мәнді қайтармайды. Екінші қиындық та біріншімен байланыста. Егер ойын ағашы өте үлкен болса, оны белгілі бір тереңдікте кесу қажет. Мысалы: *prune 0 (Node a x) = = Node a [] prune n (Node a x) = Node a (map (prune (n - 1)) x).* Мұндағы prune функциясы түбірден ***n*** қашықтықта орналасқан тармақтардың түйінінің барлығын «кесіп» тастайды. Ағаш кесілген болғандықтан, *maximise* функциясы статикалық ***n*** тереңдіктегі түйіндер үшін қолданады. Сондықтан *evaluate* былай анықтаймыз: *evaluate = = maximise . maptree static . prune 5 . gametree*; Ол, мысалы, 5 қадамға алдын ала қарайды. Біз бұл жерде жоғары деңгейлі функциялар мен «жалқау есптеулерді» қолдандық. Келтірілген *reptree* және *maptree* ойын ағаштарын оңай құрып, оларды оңай басқаруға мүмкіндік береді. Жоғарыдағы *gametree* функциясы шексіз нәтиже бере алатындықтан, бұл программа «жалқау есптеулерсіз» ешқашан аяқталмас еді. Мысалда келтірілген *(prune 5. gametree)* орнына ағаштың алдыңғы **5** деңгейін құратын бір функцияны алсақ жеткілікті болар еді. Бірақ осы бес деңгейдің өзі машина жадында көп орын алады. Біз жазған программада *(maptree static. prune 5. gametree)* функциясы maximise функциясын қажет ететін ағаш бөліктерін құрады. Құрылған бөліктер maximise функциясы онымен жұмыс істегенін аяқтаған соң, жадыдан алынып тасталып отырады (Қоқыс жинағышпен). Сондықтан ағаш толығымен жадыда үнемі болмайды. Жадыда оның тек бір кішкене ғана бөлігі болады. Сондықтан жалқау программасы тиімді. Бұл тиімділік *maximise* (тізбектегі соңғы функция) және *gametree* (бірінші), екі функцияның өзара әрекеттесуіне тікелей байланысты болғандықтан, оны жалқау есептеулерді қолданбай-ақ, бүкіл функциялардың барлығын бірге бір жерге түйіндеп, оның орнына үлкен бір функцияны қолдануға болады. Сонымен, біз әзірге минимакстік алгоритм деп аталатын тәсілдің қарапайым түрін қарап өттік. Келтірілген альфа-бета алгоритмінің негізі – бақылау, яғни бүтін ағашты қарап шықпай-ақ, *maximise* және *minimise* мәндерін есептеу [19]. Ағашты қарастырайық:

```
max
/\
min min
/\/\
1 2 0 ?
```

Ойын ағашын бағалау үшін (?) мәнін білу қажеттілігі жоқ. Сол жақтағы минимум 1-ге тең, бірақ оң жақтағы минимум 0-ден аз не тең. Сондықтан екі минимумның максимумы 1-ге тең. Бұл бақылау *maximise* және *minimise* функцияларында кіріктірілуді қажет етеді. Бірінші қадам – *minimise* сандар тізіміне қолдану: *maximise* = = *maximum . maximise'* (*minimise*, функциясында тура осындай түрлендіру болады, бірақ ол симметриялы түрде кері жағдайда өтеді). Жалқау есептеулер көмегімен егер *maximise* сандардың барлық тізімін қарастырмаса, онда олар есепке алынбайды. Көрсетілген функция анықтамасынан мынаны бақылаймыз:

maximise: maximise' (Node n []) = n : [] maximise' (Node n l) =
= map minimise l = map (minimum . minimise') l =
= map minimum (map minimise' l) =
= mapmin (map minimise' l) where mapmin = map minimum,

мұндағы *map* сандар тізімін қайтаратындықтан, олардың минимумы – *minimise* нәтижесі, онда *(map minimise' l)* сандар тізімінің тізімін қайтарады. Ал *maximise'* функциясы осы тізімдердің минумумдарының тізімін қайтарады. Бірақ бұл тізімдердегі тек максимумның мәні бар. Ол үшін *mapmin* функциясын қолданамыз:

mapmin nums : rest = min nums : omit min nums rest

мұндағы *omit* функциясы максимумының ең үлкен мәнін (оны әлеуетті максимум деп атайды) алады да, одан мәні кіші болатын кез келген минумумды алып тастайды.

omit pot [] = [] omit pot (nums : rest) =
= if minleq nums pot then omit pot rest else (minimum nums) :
: omit (minimum nums) rest

мұндағы *minleq* функциясы сандар тізімін және әлеуетті максимумды алады және егер сандар тізімінің минимумы әлеуетті максимумнан аз немесе оған тең болса, «шын» мәнін қайтарады. Егер осы шартты қанағаттандыратын элемент болса, ол тізімнің минумумы болады. Осыдан кейінгі қалған элементтердің құны шамалы болады, ол жоғарыдағы мысалда ол (?) таңбасымен белгіленген. Сондықтан *minleq* функциясы былайша анықталады:

minleq [] pot = False minleq (num:rest) pot = num <= pot || minleq rest pot

өрнектер тілі – функционалдық. Сіз ешқандай программа командасын жазбайсыз, сіз тек программаға бір ұяшықтың мазмұны басқа ұяшықтар өзара байланысқан нәтиже екенін хабарлайсыз (мысалы, «= A1 + A2»). Сіз осы кезде функционалдық программалауда программа жаздыңыз! Өте қарапайым, анық және түсінікті.

Функциялар және функцияналдық тілдердегі қосымша тиімділіктер. Функционалдық программалау тілдерінде функциялардың атқаратын рөлі зор. Функциялар мәндер рөлін де атқарады, яғни ***Int*** немесе ***String*** сияқты. Функция басқа функция нәтиже есебінде қайтара алады, ол функцияны параметр ретінде қабылдай алады және өзі де басқа функциялар көмегімен құрыла алады. Осындай жолмен программаны алу үшін, бірнеше модульдерді «жабыстыруға» болады. Мысалы, бір өрнекті есептейтін функция аргумент есебінде де жүруі мүмкін. Бұл программа модульдігін арттырады. Функцияларды басқа функциялар көмегімен құруға болатынын көрсететін мысалды қарастырайық. Ол үшін «дифференциялау» функциясын алайық. Егер бізде ***f*** функциясы болса, онда оның дифференциалын ***f'=f*** *дифференциялау* деп математикалық түрде анықтауға болатын формуланы қолдануды айтамыз.

Функциялардың мұндай түрін жоғарғы деңгейдегі функциялар деп атайды. Мысалы, Haskell тіліндегі мынандай *numOf* функцияны қарастырайық. Ол белгілі бір қасиетке ие тізімдегі элементтердің санын есептейді:

```
numOf p xs = length (filter p xs)
```

мұндағы жол қатары мынаны білдіреді: «берілген ***p*** предикатының көмегімен ***xs*** тізімін сұрыптап, нәтиже ұзындығын есептеу». Мұндағы ***p*** функциясының атқаратын қызметі элементті алып, оның тестен өтіп-өтпеуіне байланысты ***True*** немесе ***False*** мәнін қайтарады. Сондықтан *numOf* функциясы – жоғарғы деңгейдегі функция және кейбір орындалатын функциялар оған аргумент есебінде беріледі. Тағы бір атап өтетін нәрсе: жоғарыда келтірілген *filter* функциясы да – жоғарғы деңгейдегі функция, ол тестеу функциясын аргумент есебінде пайдаланады. Енді осы функцияны толығырақ қарастырайық. Одан бірнеше арнайы функцияны анықтап шығарайық.

```
numOfEven xs = numOf even xs,
```

мұндағы анықталған *numOfEven*, функциясы тізімдегі бірдей элементтер санын есептейді. Көрсетілген теңдеудегі ***xs*** параметрі анық түрде табылмайды. Біз оны *numOfEven = numOf even* деп көрсете аламыз. Енді 5 санына тең немесе одан көп болатын элементтер санын есептейтін функцияны табайық:

numOfGE5 xs = numOf (≥5) xs

мұндағы тестеу функциясы «≥5», ол қажетті функцияны қамтамасыз ету үшін оған *numOf* беріледі. Осы көрсетілген мысалдан функционалдық программалаудағы модульдік принциптің жалпы функцияларды қалай анықтайтындығын көруге болады. Мұнда функционалдық қасиеттің кейбіреуі аргумент есебінде беріліп, ал олардың көмегімен кез келген арнайы функцияларды құруға болады.

Функционалдық тілдердің негізгі идеясы мынада – функциялардың орындалу нәтижелері енгізу деректеріне тікелей байланысты болады. Haskell тіліндегі айнымалылар өзгермейді. Бұл – императивті тілді пайдаланатын программалаушылар үшін түсінуге қиын жайт. Ондай тілдерде айнымалыларға белгілі бір мәндер беріледі. Бірақ функционалдық үлгінің мән-мағынасына жеткенде, оның артықшылығын ерекше сезінуге тиістісіз. Кез келген өрнектегі есептеулерді кез келген тізбектер түрінде орындауға болады. Белгілі бір ену параметрлері енген жағдайда функция үнемі белгілі бір нәтижені беріп отырады. Бұндай анықтама императивті программаларда болып тұратын қателіктерді болдырмауға мүмкіндік береді.

Функционалдық тілдердің түсініктілігі және ондай тілдерде есепті шешудің көптеген қарапайым тәсілдерін қолдануға болатындықтан, мұндай тілдерде жазылатын программалар басқаларға қарағанда қысқа (әдеттегі программалардан 2 еседен 10 есеге дейін) болып келеді. Императивті тілдерге қарағанда, қиындықты шешу семантикасы функционалдық тілдерде анық, әрі тез шешіледі, яғни функцияның дұрыс анықталғанын табу оңай. Сонымен бірге Haskell тілі қосымша тиімділіктерді қолдануға тыйым салғандықтан, қателерді жасау да аз болады. Сондықтан Haskell тілінде жазу оңай және ол программалар тұрақты болып келеді де, оларға қызмет көрсету де жеңіл болады.

2.5. Haskell тілінің мүмкіндіктері

Қазіргі Haskell тілі – бұл жалпы мақсаттарды орындайтын қазіргі заманғы тіл. Ол – функционалдық программалауды қолдайтындардың ұжымдық білімдерін жинақтап, қуатты, жалпыға бірдей, көркем тілді құру мақсатын іске асырған тіл. Бұл тілдің өзіне қатысты бірнеше қасиеті бар. Соларды атап өтейік [17].

Тазалығы. Функционалдық программалаудың басқа тілдеріне қарағанда, Haskell тілі – таза тіл. Онда ешқандай қосымша әсерлер жоқ. Бұл – Haskell тілін ерекшелейтін қасиеттер қатарына жататын ерекшелік.

Содан соң, *maximise'* және *minimise'* функцияларын анықтап, мына жаңа *evaluate* функциясын былай деп жазуға болады:

evaluate = maximum . maximise' . maptree static . prune 8 . gametree

Жалқау есептеулер көмегімен *maximise'* функциясы ағаштың ең аз бөлігін қарастырады. Сонымен бірге әр түйіндегі ағаш тармақтарын сұрыптауға да болады. Ол үшін ең алдында бағалау бағасы ең көп түйіндерді орналастырып отырады. Оны мына функциялар көмегімен іске асырамыз:

lowfirst (Node n sub) =

= Node n (sort (map highfirst sub))) highfirst (Node n sub) =

= Node n (reverse (sort (map lowfirst sub))

Мұндағы *sort* – сұрыптаудың әмбебап функциясы. Енді мынаны анықтаймыз:

evaluate = max . maximise' . highfirst . maptree static . prune 8 . gametree

Іздеуді азайту үшін компьютердің немесе оның қарсыласының үш ең жақсы жүрісін қарастыру жеткілікті болар. Оны программалау үшін, *highfirst* мынаған ауыстырсақ жеткілікті:

(taketree 3 . highfirst)

Мұндағы *taketree n = redtree (nodett n) (:) [] nodett n x sub = = Node x (take n sub).* Аталған *taketree* функциясы ағаштағы барлық түйіндерді ***n*** түйіндерімен ауыстыратын *(take n)* функциясын пайдаланады. Бұл функция тізімнің бірінші ***n*** элементін қайтарады. Тағы қолданылатын амалдың бірі – кесуді жетілдіру. Программа алдын ала берілген тереңдікті қарап шығады. Мысалы, шахмат ойынында, егер ферзь шабуыл астында тұрса, ары қарай қарау тоқтатылады. Әдетте, үзуге тұрмайтын кейбір «динамикалық» әрекеттер алдын ала қарастырылады. Мәселен, *dynamic* функциясы prune функциясына арналған бір ғана теңдеуді қосатын әрекеттерді алдын ала біледі:

prune 0 (Node pos sub) | dynamic pos = Node pos (map (prune 0) sub).

Егер программа осылайша модульді болса, онда оған өзгерістерді оңай енгізуге болады. Жоғарыда атап өтілгендей, программа тиімділігі *maximise* және *gametree* (яғни тізбектегі бірінші және соңғы функциялар арасындағы), функциялары арасындағы әрекетке байланысты болғандықтан, жалқау есептеулерді қолданбасақ, оны бір монолитті программа есебінде жазу қажет. Мұндай программаны жазу қиындығымен қатар, оны өзгертіп түсіну тіпті де қиындық тудырады.

Жалпы жағдайда программалауда қолданылатын модульдік принцип – сәтті программалау кілті деуге болады. Модульдік программалауды іске қолдану үшін, тіл оның байланыстыру элементтерінің

типтерін қамтамасыз етуі қажет. Программалаудың функционалдық тілдерінде осындай байланыстың екі түрлі түрі берілген. Ол – жоғарғы деңгейдегі функциялар мен «жалқау есептеулер». Осы байланыстарды пайдалана отырып, бірнеше модульдерден тұратын программалар алуға болады. Көптеген кіші немесе үлкен модульдерді бірнеше қайтара пайдалануға да болады. Сондықтан функционалдық программалар көлемі жағынан аз болады және оларды жазу өте қарапайым. Программа күрделі болған жағдайдың өзінде де, жоғарыда келтірілген байланыс түрлерін аспаптық құрал есебінде пайдаланып, программаны қарапайымдауға болады.

Жалпы, «жалқау есептеулердің» болашағы зор деп есептейміз. Бұл – функционалдық программалаудың негізгі қуатты байланыс түрі ғана емес, программалау технологиясы деген деңгейге көтерілуге құқы бар есептеу түрі.

2.4. Функционалдық программалау тілдері

Абстракция деңгейлері. Жалпы жағдайда ЭЕМ программалау саласында екі түрлі аумақ бар – ресурстарды басқару және есептеу үдерісін жоспарлауды басқару. Ресурстарды басқаруда (жадыдағы регистрлерді бөлу) жадыдағы қоқыстарды жинау аспаптары бар. Бұл аспаптармен көптеген программалау тілдері қамтамасыз етілген. Есептеу үдерісін жоспарлауды басқаруда да абстракция деңгейі бар. Императивті тілдерде ол жаңа негізгі сөздер мен стандартты кітапханалар арқылы іске асады. Көптеген тілдерде цикл құруға арналған арнайы синтаксистік құрылымдар бар. Дегенмен императивті тілдерде қолданылатын технология бірінен соң бірі жазылатын программалау тәсілін пайдаланатындықтан, ондағы абстракция деңгейін көтеру жолдарының біріне қосымша негізгі сөздер мен стандартты функцияларды енгізу жатады. Мысалы, Python және Java тілдері сияқты. Ал функционалдық программалау тілдерінде дерексіздік деңгейлеріне жоғарылауға қол жеткізе алудың өзіндік жолдары бар. Мәселен, Haskell тілінде есептеу үдерісін жоспарлауды басқару жоқ. Сіздің мақсатыңыз – программа нені есептесе, соны жазасыз. Ал оны қалай, қай уақытта есептейді: оған басыңызды қатырмайсыз. Яғни Haskell тілі мәселенің өзі емес, оны шешудің бір бөлігі болады [16].

Функционалдық программалау тілдерін пайдалану. Сіз функционалдық программалау тәсілін өзіңіз оған көңіл аудармай-ақ пайдаландыңыз! Жалпы, жұрт пайдаланатын Microsoft Excel-дегі

Қалдырылған (жалқау) есептеулер. Бұл – Haskell тілінің тағы бір ерекшелігі. Ол – есептеуге қажеттілік тумаса, есептелмейді деген сөз. Мысалы, шексіз рекурсияда шексіз сандар тізімі анықталған дейік. Мұндағы пайдаланылатын тізім элементтері ғана есептеуге ілінеді. Бұл көптеген мәселелерді шешуге мүмкіндік береді. Мысалға шешімдердің барлық түрін табатын әдістер тізімін беріп, оның ішінен қажетсіздерді алып тастау сияқты жолды да атауға болады. Қалған тізімде мүмкін болатын шешімдер тізімі ғана болады. Қалдырылған есептеулер бұл операцияны өте мөлдір қыла алады. Егер сізге бір шешім қажет болса, сіз тізімнің бірінші элементін аласыз, яғни жалқау есептеулер механизмі ешқандай артық есептеулер жасамайды.

Қатаң типтеу. Атап өтсек, Haskell тілі – қатаң түрде типтелген тіл. Бұл тілде байқаусызда ***Double*** типін ***Int*** типіне келтіру мүмкін емес және бос көрсеткіш бойынша қажет әдісті де шақыра алмайсыз. Бұл қателер санын азайтуға әсерін тигізеді. Әрине, қажет болған жағдайда, мысалы, белгілі бір операцияны орындау қажеттігінде, ***Int*** типін ***Double*** типіне келтіру туындауы да мүмкін, бірақ іс жүзінде ол жиі қайталанбайды ғой. Онда бұл келтіруді ашық түрде орындау қажет. Ол үшін қиындығы бар код бөлігін бөліп қарау қажет. Басқа тілдерде, егер осындай келтірулер ашық емес түрде орындалса, онда компилятор ***Double*** типіне ***Int*** типі сияқты қатынайды немесе ***Int*** сілтемесі бар тип деп түсінеді. Басқа тілдерден бөлек Haskell тілінде типтер автоматты түрде анықталады. Сіз өзіңіз анықтаған функция типін тіпті де көрсетпей-ақ қоюыңызға болады. Кейбір код құжатында көрсетілген жағдайлардан басқа, Haskell тілі айнымалыны қалай пайдаланғаныңызды қарайды да, сіздің айнымалыңыздың қандай типте болатынын шешеді. Содан соң ол типтердің сәйкестігін өзі тексереді. Осылайша сіз қатаң типтеу артықшылығын аласыз. Және де қателер программа орындалу кезінде емес, компиляция кезінде табылып отырады. Сонымен бірге Haskell тілі көптеген жалпыға бірдей айнымалы типтеріне арналған шешімдер де қабылдайды. Мысалы, сіз типі сипатталмаған сұрыптау функциясын жазсаңыз, Haskell тілі сұрыптауға қажетті барлық мәндер үшін оның жұмыс істеуіне мүмкіндік береді. Салыстыру үшін осы операцияның орындалуын ***Java*** тілінде қарастырайық. Бұл тілде полиморфизмді орындау үшін, сіз базалық топтарды пайдаланасыз. Содан соң өз айнымалыларыңызды осы топқа жататын кіші топтың мүшесі деп атайсыз. Осының бәрі – «тоннаға» тартатын қосымша жұмыс және күрделі атауларды тудыру. Бұл көп жұмыс тек бір айнымалының бар екенін білдіруге байланыс-

ты туып отыр. Сонымен бірге типтерді ашық түрде жариялау үшін тағы да көптеген типтер түрлендірулерін атқару қажет. Егер сіз Java тілінде полиморфты программа жазғыңыз келсе, сіз пайдаланатын параметрлеріңізді ***Object*** типінде деп жариялайсыз да, функцияға оған кіруге құқы жоқ типтерді де кіргізіп жіберуіңіз мүмкін. Нәтижесінде Java тілінде жазылған функцияларда жалпылық қасиет болмайды да, олар көбінде тек бір ғана типпен жұмыс істейтін функциялар болып қала береді. Сонымен бірге бұл тілде қателер тексеру уақытын компиляция кезінде емес, орындалу кезіне ауыстырады. Көптеген жүйелерде кейбір функционалдық мүмкіндіктері сирек қолданылатындықтан, бұл қателер мүлдем көрінбей қалады. Ал кейін аталған қателер жүйенің мүлдем істен шығуына себеп болуы да мүмкін.

Ал Haskell тілі сіздің қауіпсіз әдемі, қысқа программа құруыңызға жол ашады. Құрған жүйеңізде оны істен шығарып тастайтын тығылған қателер болмайды.

Haskell тілі және қателер. Біз жоғарыда атап өткендей, Haskell тілінің ерекшеліктері қателердің етек жаюына кедергі бола алады. Енді осы ерекшеліктерді тағы да бір жинақтап атап өтейік.

1. Таза. Қосымша әсерлер жоқ.
2. Қатаң типтелген. Типтерді басқа мағынада қолдану мүлде жоқ.
3. Ыңғайлы. Программалар қысқа, ол функцияға қарап бірден «барлығын түсіну» мүмкіндігін береді.
4. Жоғары деңгейлі. Жалпы, Haskell тіліндегі программалар алгоритмді қалай сипаттаса, солай оқылады. Бұл алгоритм тұжырымдамасын функцияның анық орындап тұрғанын тексеруді оңайлатады. Әдетте, кодалау үдерісі абстракциялаудың жоғары деңгейінде өтетіндіктен, қателердің кіретін орындары азаяды. Сонымен бірге ұсақ деңгейлердегі үдерістер компилятор үлесіне тиеді.
5. Жадыны басқарады. Программадағы «іліНіп» тұрған сілтемелерге көңіл аудармай-ақ қоюға болады. Қоқыс жинағыштар оны жинақтап алады. Программалаушы тек алгоритмнің орындалуын қадағалайды да, жады көлемі жайында ойланбайды.
6. Модульді. Жалпы, Haskell тілінде алдын ала құрылған модульдерден сіздің программаңызды жасауға мүмкіндік беретін күшті механизм бар. Яғни программалар модульдерден құрала алады. Әдетте, модульдік функциялардың дұрыс орындалуын қадағалау индукция бойынша өтуі де мүмкін. Егер комбинация орындалуының дұрыстығы қадағаланса, онда олардың комбинациясы да дұрыс нәтиже береді.

Әрине, функционалдық программалау стилінде программалағанда, сіз өз мәселеңізді осы функционалдық тілде шеше бастайсыз. Сіз үлкен қиындығыңызды оңтайлы шешуге ыңғайлы кіші есептерге бөліп, осы кіші есептерді программалайсыз. Содан соң оларды біріктіресіз. Сіздің программаңызда қатеге орын да қалмайды.

Haskell тілі және ОБП (Объекті бағытталған программалау). Нысан бағытталған программалау тілдерінің маңызды ерекшеліктерінің біріне деректерді оларға әсер ететін функциялар көмегімен топтау жатады. Бұл функция нысанға әсер етеді. ОБП бұл қасиеттерін *деректер инкапсуляциясы* (интерфейс пен іске асыруды бөліп қарау) және *полиморфизм* (бірдей типтегі деректер жиынтығының бірдей тәртіптегі қозғалысы) деп атайды. Бірақ инкапсуляция мен полиморфизм қасиеті ОБП-дан да басқа программалау стильдерінде бар. Жалпы, Haskell тілінде деректерді абстракциялауға арналған құрал-аспаптар бар. Деректерді инкапсуляциялау Haskell тілінде жеке модульде әр дерекке қатысты типін анықтаудан басталады. Бұл модульден сіз тек интерфейсті сыртқа шығарасыз. Модуль ішінде деректермен әрекеттесетін функциялар жиынтығы болуы мүмкін. Бірақ модуль сыртында тек интерфейс көрініп тұрады. Полиморфизм, әдетте, кластар типін пайдаланудан құрылады. Егер сіз С++ немесе Java тілінде программаласаңыз, онда кластарды нысандар шаблоны сияқты ұқсастырып жасайсыз. Бірақ Haskell тілінде кластар типін тура мағынасында түсіну қажет. Бұл тілде класс типке кіре ме, жоқ па, оны ережелер жиыны анықтайды. Осылайша Haskell тілі класс жеке даналары мен деректер типін құруды бөліп қарайды. Мысалы, сіз «Порше» типін «Машина» типінің жеке данасы ретінде анықтауыңызға болады. Осы типке қолданылатын барлық функциялар «Порше» типіне де қолданылады. Мысалы, Haskell тіліндегі типтер ***Show*** класының данасы бола алады. Ол деректер типін жол қатарына аударатын *show* функциясы көмегімен іске асады. Демек, Haskell тіліндегі барлық типтерді *show* функциясын қолданып, оны жол қатары ретінде экранға шығаруға болады. Haskell тілінің жүйесі полиморфизм ұғымдарымен операция жасауға ыңғайлы. Сіз мұрагерлік құрылымның иерархиялық жасалуын қадағаламайсыз, сіз тіпті иерархия бар ма, жеке даналарма оны да ойланбайсыз. Мұнда Кластың өзі клас, ал типтер осы кластың экземплярлары, және аталық-балалық қатынас түрін сақтап тұру міндетті емес. Егер деректердің типі класс талаптарына сай келсе, онда ол осы кластың экземпляры болады. Өте қарапайым және түсінікті. Мысалы, *QuickSort* функциясын алайық. Мұндағы полиморфизм құпиясы мынада: ол *qsort* функциясында. Бұл функция *Ord*

класының кез келген типімен жұмыс істеуге бағытталған. Аталған *Ord* класында белгілі бір функциялар жиыны бар. Оның ішінде «<» және «≥» амалдары бар. Егер біз *Ord* функциясын біздің «Порше» типі үшін анықтасақ (мысалы, мына амалдарды қолдану керек дейік, «≤» немесе «==», одан ары қарай Haskell тілі өзі анықтайды), біз сұрыпталған «Порше» тізімін qsort функциясы көмегімен анықтаймыз. Бұл жерде тізім элементтерінің қай класқа жататыны анықталмайды: оны Haskell тілі автоматты түрде өзі анықтайды. Сол үшін ол қандай функцияны пайдаланғанды қарап шығады. Мысалы, *qsort* функциясы үшін екі амал: «<» және «≥» қарастырылады. Сонымен, қорыта айтсақ, Haskell тілінің құрамына деректерге инкапсуляция жасайтын механизм кіреді. Бұл механизмдер өзі мүмкіндігі жағынан ОБП механизмінен асып түсуі де мүмкін. Haskell тіліндегі жоқ нәрсе – ол функцияларды нысандарға топтау тәсілі. Есіңізде болар, функцияның өзі деректер болып та қызмет атқарады. Бұл аталған кемшілікті жою жолдары да бар. Мысалы, функцияны нысанға қолдану үшін *«func obj a b c»* деп жазсақ болады [20].

Модульдігі. Компьютерлік ғылымдарда «модульдік» ұғымы негізгі ұғымдар қатарына жатады. Өнімділікті арттыруға мұқтаж программалау тілдері модульдік программалауды міндетті түрде қолдауы қажет. Ол үшін шолу жасайтын жаңа ережелер жиыны болуы және компиляция режимінің бөлектігін қамтамасыз ететін механизмнің болуы жеткілікті емес. Модульдіктің болуы тек қана модульдер болуымен шектелмейді. Берілген мәселеге қатысты есепті шешуде оларды кіші бөліктерге бөлу, соңынан оларды қайтадан бір бүтінге жинақтау байланыстары болуы керек. Модульдік программалаудың болуын қамтамасыз ету үшін программалау тілінде өте жақсы байланыстырушы элемент болуы қажет. Функционалдық программалау тілдері осындай байланыстарды қамтамасыз ететін элементтердің екі түрлі түрін бере алады. Ол – жоғары деңгейдегі функциялар және қалдырылған (жалқау) есептеулер.

Haskell тілінің шапшаңдығы. Программалау тілдерінде шапшандықтың атқаратын белгілі бір рөлі бар. Егер сіз құратын программа үшін шапшаңдық белгілі бір рөл атқарса, онда, әрине, оны есепке алу қажет. Мысалы, қазіргі кезде C++ тілінде жазатын программалаушылар жоғарыда келтірілген *QuickSort* функциясы C++ тілінде шапшан орындалады деген сенімде. Ол шындық болуы да мүмкін. Кейбір алгоритмдер шапшаңдығы программалау тілдеріне де байланысты болады. Мысалы, тізімдерді сұрыптау алгоритмдерінің қайсысы шапшаң жұмыс істейді деген сұрақты да қоюға болады. Кейбір қысу алгоритмі

осы жұмысты атқаруы мүмкін. Егер сізге шын мәнінде шапшаңдық қажет болса, онда программаны C++ тілінде де, Haskell тілінде де құрыңыз. Содан соң оларды өзара салыстырыңыз.

Компьютерде программалаудың бұрынғы заманнан келе жатқан мынандай ережесі бар: *«80/20 ережесі»*. Бұл ереже бойынша 80% уақыт 20% программа кодасын жазуға кетеді. Бұл ереженің көлемі жағынан аз болатын да, көп болатын да программа модульдеріне қатысы бар. Бірақ құрылған программа қолданбасы үлкейген сайын 80% тік бөлігі арта береді де, 20% тік бөлігі азаяды. Яғни программа модульдері «берілген амалдарды жиі орындауға» емес, «іс-әрекетті көбірек орындауға» бет бұрады. Бұл жерде мына ереженің орындалуы дұрыс болар еді: «30% уақыт 1% коданы жазуға, ал қалған 70% уақыт қалған 99% тағы да кодаға кетеді». Әдетте, қолданбалар өз функциалдығын арттыруға күш салады, сондықтан көптеген функциялардағы өңдеу уақыты көбейеді. Қазіргі кезде қолданба программалар Ассемблер не Си тілінде жазылмайды. Алгоритм түрлерін онтайландыру программа кодасын оңтайландырудан жақсы және сапалы нәтижеге жеткізеді. Егер қолданба жазғанда жазу уақыты мен тұрақтылық факторлары аса маңызды болмаса, әрине, Си тілі Haskell тілінен гөрі тиімді болады. Бірақ өмірде қолданба жазу үдерісі қымбат үдерістерге жататындықтан, Haskell тілінде жазылатын программалар Си тіліне қарағанда 10 рет тез жүреді, сондықтан сіз қалған уақытыңызды алгоритмді оңтайландыруға жұмсауыңызға болады.

Енді біз мынандай сұраққа келдік. Егер Haskell тілі сондай керемет болса, онда ол неге жалпыға кеңінен тарамады? Оның себептерінің біріне операциялық жүйе Си тілінде жазылғандықтан, сіздің, әрине, өз қолданбаңызды осы ортаның тілінде жазғаныңыз тиімді әрі пайдалы болады. Сонымен бірге Haskell тілі мен басқа да функционалдық программалау тілдерінің кең тарамауы себептерінің біріне адамдар арасында қалыптасқан белгілі бір пікірлер жүйесі жатады. Сондай пікірлердің біріне программалау тілдерін адамдардың құрал деп есептемеуі жатады. Көптеген программалаушылар үшін олардың үйренген программалау тілдері олардың діндері сияқты. Олардың оны өзгерткісі келмейді.

2.6. Haskell тілінің негіздері

Қазір мектептердің өзінде программалаудың Паскаль, Си немесе Java тілдері үйретіледі. Бұл тілдердің барлығы *программалаудың императивті тілдеріне* жатады. Оларда алгоритмдер белгілі бір

іс-әрекеттердің тізбектері ретінде берілген. Программалаудың басқа стиліне немесе түріне *функционалдық программалау* жатады. Функционалдық программалаудағы программаларда нені есептеу қажет екені анықталады да, ал орындау тізбегі жасырын түрде қалады. Оны программалаушылардың орындалған арманы деуге де болады: «Мен өзіме *не* қажет екенін сипаттаймын, ал компьютер оны *қалай* жасау керек екенін өзі шешуі керек» [21].

Сонымен, Haskell программалау тілі – бұл «жалқау» типтердің полиморфтық қасиеті бар функционалдық программалау тілі. Haskell тілі функционалдық тіл болғандықтан, оның негізгі ұғымы – функция. Бірақ функциялар барлық тілдерде бар ғой. Бұл жерде программалау стилі жайында сөз болмақ. Функционалдық программалау тілдерінде функциялармен жұмыс атқарғанда, оларды сандар есебінде де, жолдық қатар айнымалылары есебінде де санауға болатынын атап өткенбіз. Мысалы, бір функция аргумент есебінде басқа бір функцияны қабылдап, үшінші функцияны нәтиже есебінде қайтарады. Функционалдық программалаушы функциялар және бір-біріне тәуелді функциялар терминіне қатысты мәселені шешсе, ал императивті программалаушы іс-әрекет және осы әрекеттерді процедуралар тізбегі терминдерде ой жүргізеді. Функционалдық программалау тілінде программа жазу – есептеуге қажетті өрнекті жазып, осы өрнекте пайдаланылатын функцияны сипаттау деген сөз. Бұл жерде негізгі көңіл аударатын нәрсе – қандай тізбекпен қандай командалар орындалатыны емес, көрсетілетін функциялардың бірінің бірі арқылы өрнектелуіне және соңында қандай нәтиже алынатыны. Функционалдық программалаушы өз программасының іс-әрекеті жайында ойланбайды да. Функционалдық ойлау деңгейі әдетте кестелік процессорларда кеңінен сипатталады. Біз ұяшықтардың бір-біріне тәуелділігінің сипатын ғана анықтаймыз да, басқа ұяшықта нәтиже аламыз. Haskell тіліндегі программа бір өрнекті сипаттайды. Онда екі бөлікті бөліп қарауға болады: *where* («қайда» деп аударылады) сөзіне дейін және одан кейін [22]. Мысалы, мына про-грамма:

```
1 + (x + y) * 2 where x = 1
y = 2
```

Нәтиже есебінде 7 санын қайтарады. Өрнектегі қолданылатын функциялар *where* сөзінен кейін анықталады.

Haskell тілінде айнымалылар жоқ. Бұл – Haskell тілін «таза» деген ұғымға жатқызатын қасиеттердің бірі. Айнымалылар жоқ, бірақ

оны аргументті қабылдамайтын және санды қайтаратын функция арқылы анықтауға болады. Мәселен, жоғарыда көрсетілген мысалдағы ***x*** және ***y*** таңбалары айнымалыларды емес, функцияларды білдіреді. Haskell тіліндегі «=» таңбасы Си тіліндегі (немесе Паскаль тіліндегі := операциясы) осындай таңбамен берілетін меншіктеу операциясынан бөлек мағынада қолданылады. Си тілінде бұл операция былайша таратылады: «тең» таңбасының оң жағында орналасқан есептеулерді орындап, нәтижені осы таңбаның сол жағында орналасқан айнымалыға (жады ұяшығына) орналастыру. Көрсетілген мына қатар: ***x = x + 2*** Си тілінде былайша түсіндіріледі: «***x*** айнымалысының мәнін 2-ге арттыру». Ал Haskell тілінде бұл команда мағынасы мүлде бөлек. Ол мынаны білдіреді: «аталған ***x*** функциясын қолдану арқылы функция нәтижесі мынаған: **2** саны мен ***x*** функциясының есептеу нәтижесінің қосындысына тең болады». *Яғни* Haskell тілінде бұл жазылған жол қатары ***x!!!*** деген атауы бар рекурсивті функцияның анықтамасы болып табылады. Көрсетілген ***x*** функциясы өзі арқылы анықталған. Және осы функцияны пайдалану шексіз рекурсивті шақыруға келтіріп, ең соңында стектің толып кету қатесіне әкеледі. *Яғни «stack overflow»*:

```
> x where x = x + 2
ERROR – stack overflow
```

Haskell тілінде айнымалылар жоқ болғандықтан, «күй» ұғымы да жоқ. Бұл ұғымға ағымдағы барлық айнымалылар мәндерінің жиындары жатады. Мұндай қатаң және әдеттен тыс жағдайларға қалай шыдауға болады? Бірнеше қарапайым мысал қарастырайық. Бұл программалау тілінде мынандай базалық типтер бар: ***Integer*** (бүтін сан), ***Char*** (таңба), ***Double*** (жылжымалы үтірі бар сан, ол бұрын ***Float*** деп белгіленген). Сонымен бірге мынандай арнайы конструкциялар **()**, **[]** және →, бар. Олар бар типтер негізінде жаңа типтерді анықтайды [23].

Деректердің белгілі бір типтері ***a*** және ***b*** берілген дейік. Сонда мынандай конструкция ***[a]*** жаңа типті анықтайды. Ол ***a*** элементтері типіндегі тізім болып табылады. Атап айтсақ, ***String*** деген тип ***[Char]*** типінің синонимі бола алады. Көрсетілген ***(a, b)*** конструкциясы мынаны: ол ***a*** және ***b*** типіндегі элементтер жұбын білдіреді. Осылайша үштік, төрттік және көптеген жиынтықты ***n*** элементтерден тұратын жиындарды (кортеждерді) алуға болады. Көрсетілген ***(a, b)*** конструкциясы мына функцияға сәйкес келеді: ол енуде ***a*** типіндегі элеметті алып, ал шығуда ***b*** типіндегі элементті қайтарады.

Типтер мысалдары:

Integer -> Integer	бүтін аргументтің бүтін санды функциясы;
[Integer] -> Float	бүтін сандар тізімін алып, *Float* типіндегі нақты санды қайтаратын функция;
Float -> Float -> Float	енуде екі нақты санды алып, нақты санды қайтаратын функция;
(Float, Integer) -> [(Float, Float)]	енуде *Float* және *Integer* типіндегі екі нақты санды алып, *Float* типіндегі сандар жұбының тізімін қайтаратын функция.

Ерекше қызығушылық тудыратын тип түріне *Float→Float→ → Float* типі жатады. Оны былайша талдауға болады: ***Float*** типіндегі бір санды алып, *Float→Float* типіндегі функцияны қайтаратын функция. Шын мәнінде, егер екі аргументті функцияға бір аргументті меншіктесек, онда бір аргументті функцияны аламыз. Типтерді құрастырғанда бөлінбейтін элементтерді белгілеу үшін, дөңгелек жақшаларды қолданамыз. Мысалы, *(Float→Float)→Float* типі енуде *Float→Float* типіндегі функцияны алып, нақты сандарды қайтарады. Мына жағдайды ерекше атап өтейік. Жаңа типтерді ***[a], (a, b)*** және ***a→ b*** операциялар көмегімен құрғанда, ***a*** және ***b*** орнына нақты бар типтерді қою міндетті емес. Ол үшін кез келген тип түрін білдіретін кіші латын әріптерін қолдансақ болады. Мысалы ***a→b→[(a, b)]*** енуде ***a*** және ***b*** типіндегі элементтерді алып, шығуда ***a*** және ***b*** типіндегі элементтер жұбының тізімін қайтаратын типтегі функцияны білдіреді.

1-мысал. Мына *inc* функциясы кез келген санды 1 санына арттырады. Ол былайша анықталады:

inc n = n + 1 – a→a типіндегі функция.

Haskell тіліндегі түсіндірмелер екі дефистен басталады. Мына өрнек *inc (inc 3) 5* санына дейін редукцияланады немесе есептеледі деуге болады. Бұл былайша жазылады:

inc (inc 3) → 5

Функция анықтамасының алдына бір қатарды қою арқылы функцияны типтеуге мүмкіндік аламыз.

inc :: Integer→Integer

2-мысал. Мына *add* функциясы берілген сандар қосындысын табады. Бұл функцияның аргументі болып екі сан табылады:

```
add :: Integer→Integer→Integer
add 3 5 where add x y = x + y
```

Көрсетілген *inc* функциясын *add* функциясы арқылы анықтауға болады:

```
inc = add 1,
```

мұндағы *add* функциясындағы бір аргументті тұрақтандыру арқылы бір аргументті функцияны аламыз. Мынаны байқаған шығарсыз: аргументтерді қоршайтын дөңгелек жақшалар қолданылмайды. Аргументтер Си немесе Паскаль тіліндегідей үтірмен емес бос орынмен бөлектенеді.

3-мысал. Тізімдерді белгілейтін екі түрлі тәсіл бар. Бірі – төртбұрыш жақшалар, онда үтір немесе дөңгелек жақшалар арқылы элементтер сипатталады. Мысалы, мына жазба *[1, 2, 3], (1:2:3:[])* және мына жазба *(1:(2:(3:([]))))* екеуі бірдей. Қос нүкте таңбасы «:» мынаны: тізімге элементті сол жағынан қосақтау операциясын білдіреді. Екінші тәсіл. Кез келген ***x*** элементі бар дейік, ал ***xs*** осы ***x*** типтегі элементтер тізімі дейік. Сонда ***x:xs*** өрнегі тізім болып табылады. Ол ***xs*** тізімінен ***x*** элементін тізім басына орналастырғаннан пайда болады. Мына қызықты қараңыз: ***(x:xs)*** конструкциясын «тең» таңбасының сол жағынан да пайдалануға болады. Онда оған тізімнің бірінші элементін алып тастау операциясы қолданылады. Бұл *length* белгісіз типтегі элементтерден тұратын тізім ұзындығын табуға мүмкіндік беретін функцияны рекурсивті түрде анықтайды:

```
length :: [a] → Integer
length [] = 0
length (x:xs) = 1 + length xs.
```

Мұндағы *length [] = 0* катары мынаны білдіреді: бос тізім ұзындығы 0-ге тең. Жоғарыдағы көрсетілген үшінші катардағы жол, яғни *length (x:xs) = 1 + length xs* мынаны білдіреді: тізімнен бір элементті алып тастағанда, оның ұзындығы 1 санына тізімнің қалған ұзындығын қосқанмен тең.

4-мысал. Мынандай *map* фунциясын қарастырайық: ол өз енуінде функция және элементтер тізімін алады да, нәтижесінде элементтер тізімін қайтарады. Ол тізімге мына функцияны қолданамыз:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Мұндағы «тең» белгісінің сол жағындағы «:» – таңбасы «жұлып алу» дегенді білдірсе, ал оң жағындағы – «жапсыру» мағынасын білдіреді. Мысалы, мына өрнекті:

map (add 10) [1, 2, 3] [11, 12, 13]

Былайша талдауға болады: «көрсетілген тізімнің **[1, 2, 3]** әр элементіне 10 санын қосу». Бұл жердегі атап өтетін қызғылықты жағдай – функцияларды өрнектің өзінде бірден құрастыруға болады. Аргументті тұрақтандыру үшін «\» таңбасы қолданылады. Мәселен, мына өрнек $\backslash\, x \rightarrow x * x * x$ мағынасын былайша түсіндіруге болады: берілгені мына функция $f(x) = x^3$, ал аталған өрнек $(\backslash\, x \rightarrow x * x * x)\ 3$ мәні 27 тең болады.

map (\ x → x * x * x) [1, 2, 3] → [1, 8, 27]

Көрсетілген map функциясын анықтаудың тағы бір тәсілі бар:

map f xs = [f x | x ← xs]

Көрсетілген «|» белгісі мына операцияны анықтайды: бір функция есептеуінің нәтижесі екінші функцияға ену болып табылады. Бұл мынандай математикалық жазбаға сәйкес келеді:

map (f, xs) = {f(x) | x ∈ xs]

Мұндағы $x \in xs$ өрнегін «генератор» деп атайды, оны былайша таратуға болады: «аталған ***x*** элементі ***xs*** тізімінен алынады».

Шексіз тізбектермен жұмыс істеу. Haskell тілінде шексіз нысандармен жұмыс істеуге мүмкіндік бар. Ол үшін, мысалы, *натурал сандардың* шексіз тізбегін немесе *Фибоначчи сандарының* шексіз тізбегін немесе басқа да бір шексіз тізбектерді нәтиже есебінде қайтаратын функцияны анықтасақ жеткілікті. Мысалы, мына өрнек:

ones = 1 : ones

конструкцияда көрсетілгендей, *ones* функциясы бірлерден тұратын шексіз тізбектерді анықтайды. Шынында да, егер біз осы рекурсивтік анықтауды одан ары қарай шеше бастасақ, онда мынандай өрнекті аламыз:

ones = 1 : 1 : ones
ones = 1 : 1 : 1 : ones
ones = 1 : 1 : 1 : 1 : ones
...

Бұл тізбек 1 санын тізбек басына қосқаннан соң, өзіне-өзі тең болып қала береді. Жалқау емес тілдер олардан сұраған нәрсені бірден орындағысы келгендіктен, олар тез арада орындау қателерін жасай бастайды (мысалы, жадының толып кету қатесі сияқты). Ал «жалқау», өзінің «тең» белгісінің оң жағындағы анықтамасын шешуге асықпайды, ол оны қажеттілік туғанда ғана ашып отырады. Мұндай «тең» белгісін «жалқау тең» белгісі деп атайды, ол шын мәнінде меншіктеу операциясын емес, функция анықтамасын білдіреді. Функционалдық тілдерде

осындай «тең» белгісінің екі түрі бар. Бірі «жалқау» – ол функцияны анықтау. Екіншісі «жалқау емес» – белгінің оң жағындағы өрнекті есептеу және «тең» белгісінің сол жағындағы айнымалыға нәтижені меншіктеу. Мысалы, мынандай *Mathematica* функцияны анықтау үшін, аталған тіл «:=» таңбасы, ал меншіктеу үшін «=» таңбасы пайдаланылады. Бір аргументті ***n*** – бүтін санын қабылдайтын *numsFrom* функциясын қарастырайық. Ол функция ***n*** үлкен немесе оған тең барлық бүтін сандар тізімін қайтарады.

```
numsFrom n = n : numsFrom (n + 1)
```

Аталған функцияны пайдалану арқылы бүтін сандар квадраттарының шексіз тізбегін аламыз:

```
squares = map (^2) (numsFrom 0)
```

мұндағы (^2) өрнегі – берілген санды квадрат дәрежесіне шығаратын функция. Аталған тізбектің алғашқы элементтерін *take* функциясының көмегімен алуға болады:

```
take 5 squares → [0, 1, 4, 9, 16]
```

Көрсетілген *take* функциясын рекурсивті түрде былайша анықтаймыз:

```
take :: Integer →[a]→[b]
take 1 (x:xs) = [x]
take n (x:xs) = x : take (n – 1) xs.
```

Ал мынау – 2 санының бірінші 5 дәрежесін анықтаудың қарапайым тәсілдерінің бірі:

```
take 5 powers where powers = map (2 ^) [1..]→[2, 4, 8, 16, 32].
```

Санды екінің дәрежесіне жіктеу есебі [21]. Жалпы жағдайда нүкте көмегімен анықталатын екі түрлі функция композициясынан тұратын арнайы операция бар. Осындай композиция былайша: $h(x) = f(g(x))$ анықталса, онда ол Haskell тілінде былайша жазылады:

```
h = f . g
```

Көрсетілген композиция операциясы Haskell тілінде кәдімгі қарапайым функция ретінде анықталады:

```
(.) :: (b→c)→(a→b)→(a→c)
f . g = \x → f (g x).
```

Енуде екі функция алып, шығуда бір функцияны қайтарады. Композиция операциясын пайдалану арқылы *toDigits* функциясын анықтаймыз. Бұл берілген функция бүтін сан үшін екілік түрдегі разрядтар тізімін табады. Сонымен қатар *countUnits* функциясын анықтайық. Ол функция натурал сандардың екілік жазбасындағы бірлер санын есептейді. Мысалы, $n = 11$ саны үшін аталған *countUnits* функциясы 3 санын қайтаруы керек. Өйткені $11 = 1 + 2 + 8 = 2^0 + 2^1 + 2^3$,

ал 255 саны үшін жауап 8 болуы керек, өйткені 255 = 1 + 2 + 4 + ... + + 64 + 128 дегеніміз – екі санының әр түрлі дәрежесінің 8 рет қолданғандағы нәтижесі. Мынандай анықтамаларды енгізейік.

```
toDigsI :: Integer -> [Integer]
toDigsI n | n == 0 = []
| otherwise = (n `mod` 2) : toDigsI (n `div` 2)
countUnits = sum . toDigsI
toDigits = reverse . toDigsI
```

Мұндағы *toDigsI* функциясы мынаны: *n* санының екілік көрсетіліміндегі разрядтарының тізімін оңнан сол жақ бағытқа қарай анықтайды. Мысалда көрсетілген *reverse* функциясы тізімді тізімге қайтарады, яғни енуде бір тізімді алып, шығуда екінші тізім қайтарады. Қайтарған тізімдегі элементтер кері бағытта жазылады, яғни соңғы элементтен бастап, бірінші элементке дейін:

```
toDigsI (1 + 8 + 16)       →    [1, 0, 0, 1, 1]
reverse [1, 0, 0, 1, 1]    →    [1, 1, 0, 0, 1]
toDigits (1 + 8 + 16)      →    [1, 1, 0, 0, 1]
countUnits (1 + 8 + 16)    →    3
countUnits 255             →    8
```

Сан екі санының дәрежелері бар қосындыдан тұрады. Екі санының дәрежесін табу үшін, мына *zipWith (*)* операциясын колданамыз. Ол операция екі тізім элементтерінің көбейтіндісінен тұрады. Атап айтсақ, екілік разрядтар жіктелуінің тізімі мен екі дәрежесінің тізімінен тұрады:

```
powers = map (2 ^ ) [0..]
toPowers' = (zipWith (*) powers) . toDigsI
```

Жалпы алғанда, *zipWith (*)* $[a_1, a_2, a_3, ...]$ $[b_1, b_2, b_3, ...]$ өрнегі мынаған тең $[a_1*b_1, a_2*b_2, ...]$, мұнда жұлдызша орнында кез келген операция тұруы мүмкін. Жекелеген жағдайда *zipWith (+) [1, 2, 3] [10, 100, 1000]* өрнегі *[11, 102, 1003]* нәтижесін береді. Ең соңында біз мынаны аламыз:

```
toPowers' (16 + 8 + 1) [1, 0, 0, 8, 16]
```

Енді қалған жағдайда нөлдік элементтерді сүзудің қадамын қосу ғана қалды. Оның екі түрлі тәсілі бар. Бірінші тәсіл:

```
toPowers = (filter (/=0)) . (zipWith (*) powers) . toDigsI
```

Мұндағы *filter* функциясы енуде буль функциясын («шын» немесе «өтірік» мәндерін қайтаратын функция) және тізімді алады да, ал шығуда осы аталған функция элементтерінің мәні «шын» мағынасын қабылдайтын тізімді қайтарады. Бұл жағдайда біз тек нөлдік емес элементтерді қалдырамыз:

```
toPowers (16 + 8 + 1) → [1, 8, 16]
```

Сонымен, *zipWith* функциясы үш аргумент алады. Егер тек екі аргумент көрсетілсе, онда ол бір аргументті функция болады. Бұл мына төменде көрсетілген өрнекті *(zipWith (*) powers)* бір аргументті функция ретінде қолданып, оны *toDigsI* функциясымен бірге бір композиция тізбегіне орналастыруға болатынын көрсетеді. Бұл жағдай *filter* функциясына да ұқсас: біз оған бірінші аргументті береміз – **(/=0)** – бұл нөлмен салыстыратын функция. Екінші аргумент анықталмаған. Ол оған тізбек бойынша *(zipWith (*) powers)* функциясының мәні ретінде беріледі де, пайдаланушы *toPowers* функциясының аргументі ретінде *toDigsI* функциясына еніп, соның қайтарымын алады. Аталған *toPowers* функциясындағы нүктелер стандартты Linux қоршамасындағы «|» (pipe) операциялар рөлін атқарады. Бұл операция көмегімен бір функция есептеулерінің нәтижесі екінші функцияға ену есебінде кіруі іске асырылады. Жоғарыда көрсетілген мысалда үш функциядан тұратын тізбек көрсетілген.

Аталған *toPowers* функциясын басқаша да анықтауға болады. Ол үшін екінші түрлі тәсілді қолданамыз.

```
toPowers = \n -> filter (/= 0) (zipWith (*) powers (toDigsI n))
```

Ал үшінші тәсілде:

```
toPowers n = filter (/= 0) (zipWith (*) powers (toDigsI n))
```

функциясы қолданылады. Бұл көрсетілген тәсілдерде біз *n* аргументін ашық түрде енгіземіз және оны өрнекте пайдаланамыз. Бұл жағдайда жақшаларды басқаша түрде қолданамыз.

Егер f_1, f_2, f_3 өрнегі функция болса, онда олардың композициясы болып табылатын h функциясын былай анықтауға болады: $(h(x) = f1(f2(f3(x))))$. Оны Haskell тілінде үш түрде анықтаймыз:

```
h = f1 . f2 . f3
h x = f1 (f2 (f3 x))
h = \x → f1 (f2 (f3 x))
```

Тез сұрыптау. Осының алдында біз қарапайым мысалдарды қарастырдық. Енді күрделі алгоритмдер қатарына жататын *Хоардың тез сұрыптауы* деген алгоритмді қарастырайық [16]. Күрделі деген атына қарамастан, оның сыртқы түрі өте қарапайым:

```
qsort [] = []
qsort (x:xs) = qsort [y | y ← xs, y < x] ++ [x] ++ qsort [y | y ← xs, y ≥ x].
```

Мұндағы мына жазбаның *qsort [] = []* мағынасы мынандай: егер енуге **[]** – бос тізімі берілсе, онда шығу нәтижесі де бос тізім болады. Келесі қатар мына жағдайды сипаттайды: тізім бос емес және одан ***x*** бірінші элементін алуға болады. Тізімнің қалған бөлігі ***xs***

деп белгіленген. Келесі көрсетілген өрнек $[y \mid y \leftarrow xs,\ y < x]$ мынаны білдіреді: x элементінен қатаң түрде кем болатын xs тізімінің элементтерінің жиынына тең болады. Ал мына көрсетілген өрнек $[y \mid y \leftarrow xs,\ y \geq x]$ x элементінен үлкен не тең болатын xs тізімінің элементтерінің жиынына тең болады. Бұдан ары қарай біз бұл екі тізімді *qsort* функция көмегімен сұрыптаймыз және үш тізімді жабыстырамыз: Оларға *qsort* $[y \mid y \leftarrow xs,\ y < x]$ тізімі, бір элементтік $[x]$ тізімі және *qsort* $[y \mid y \leftarrow xs,\ y \geq x]$ тізімі жатады.

Осы алгоритмді Си тілінде (тек бүтін сандар үшін) жазу үшін, көбірек кодалауды орындау қажет:

```
void qsort(int *ds, int *de, int *ss){ ;
int vl = *ds, *now = ds + 1, *inl = ss, *ing = ss + (de – ds);
if(de <= ds + 1) return;
for(; now != de; ++now){ ;
if(*now <= vl) *inl++ = *now;
else *ing-- = *now;
}
*++inl = vl;
qsort(ds, ds + (inl - ss), ss);
qsort(ds + (inl - ss), de, inl + 1);
}
```

Осылайша Haskell тілінде көптеген алгоритмдерді кез келген императивті тілдерге қарағанда, мысалы, (Си, Паскаль) қысқа түрде кодалап жазуға болады [22].

Haskell тілінде өздеріңіз программа жазып үйренгіңіз келсе, әдебиетте келтірілген сайттар мен жазылған кітаптар көмегімен программаны құрып жазуыңызға болады [15-23].

Haskell дистрибутивтері Жалпы Haskell тілінің екі интерпретаторы бар. Бірі *Windows*-қа арналған, екіншісі, *Linux*-ке арналған. Олардың барлығы да ақысыз. Ең оңтайлысы қарапайым *HUGS* деп аталатын интерпретатордан бастаған жөн [18].

2.7. Лисп тілінің негіздері. Лисп тіліндегі рекурсия

Бұл параграфта Лисп тіліндегі таңбалар мен тізімдер олардың логикалық мәндері, Лисп тіліндегі қарапайым рекурсия, Трассировканы пайдалану, Өзара байланысқан рекурсия, Жоғары дәрежедегі функциялар туралы мағлұматтар келтірілген. Сонымен бірге лямбда-есептеулер редукция тәртібі және қалыпты формалар жайында негізгі

түсініктемелер берілген. Лисп тіліндегі рекурсивттік өрнектер, функционалдар туралы түсініктемелер берілген [11].

Лисп тілін ең алғаш рет Джон Маккарти (John McCarty) 1950 жылдың аяғында ұсынған. Ең алғашында бұл тіл рекурсивтік функциялар теориясына негізделген есептеу үлгісі ретінде қарастырылды. Өзінің ең бірінші еңбегінде Маккарти алдына қойған мақсатын былайша бейнелейді: рекурсивтік функциялар теориясына негізделген есептеу үлгісін іске асыратын тіл құру керек. Бұл тіл сандық есептеулер емес, таңбалармен жұмыс істеуі керек. Сонымен қатар тілдің анық түрде сипатталған синтаксисі мен семантикасы және тіл есептеу үлгісінің толық түрде шығатынын бейнелеп көрсете білуі қажет. Лисп тілі Фортран және Кобол тілдері сияқты көне тілдер қатарына жатқанымен, ол үнемі өз дамуында программалау тілдерінің алдында келеді. Функционалдық және логикалық принциптерін қолдана отырып, программалаудың бұл үлгісінің негіздерін басқа тілдер (SCHEME, ML и FP) пайдаланып отыр [13].

Лисп тілі – жасанды зерде тәсілімен құрылатын программалар үшін ең тиімді де қажет құрал. Лисп тілінің трансляторының бірнеше түрі бар. Бұл тілдің ерекшелігі – жаңа ұғымдар құрудағы тиімділігі. Жаңадан пайда болған программаның нысаны есебінде басқа программалар болуы мүмкін, сырт көзге бұл программалар мәліметтерінде ешқандай ерекшелігі жоқтай көрінеді. Бұл басқа тілдерде жоқ мүмкіндіктерді ашып береді, яғни бұл тілдің көмегімен сырткы әсердің арқасында өзгеріп отыратын программа жасау мүмкіндігі туады. Лисп тілінде машина жады былайша пайдаланылады: егер жаңа ұғымды жадыға орналастыру жағдайы кажет болғанда, машина зердесінен осы ұғымға қанша орын керек болса, сонша беріледі. Мұнда Паскаль тіліндегідей қанша орын қажеттігі алдын ала дайындалмайды. Керісінше, ұғымды өшірген жағдайда оның алып тұрған орны бірден босатылады. Лисп тілінің тағы бір ерекшелігі – программаларды машина жадында ұстау тәртібі. Мұнда олар бірінің ішіне бірі кірген түрінде сақталады. Бұл іздеуге өте оңай және аз орын алады. Соңғы кезде Лисп тіліне тиімді программалар жасау үшін ыңғайлы болатын бірнеше қосымша қосылған. Жасанды зерде бағытындағы көптеген программалар осы Лисп тілінде жазылады [12].

Лисп тілінің басқа программалау тілдерінен негізгі айырмашылығы қатарына мыналар жатады:

1) деректерді сипаттайтын негізгі құрылым ретінде тізім қолданылады;

2) бұл тілдегі программалар құрылымы да тізімдік түрде болады;
3) тілдегі негізгі орындалатын операциялар, тізімге арналған операциялар.

Лисп тіліндегі деректерді сипаттайтын негізгі блок ретінде таңбалық өрнектер қарастырылады. Қарапайым түрдегі таңбалық өрнектер әріптен басталады, құрылымда әріптер мен цифрлар бар. Жолдар атомдардан тұрады. Тілдің ішкі құрылымында атом жады ұяшығымен беріледі. Т таңбасы жеке атом түрінде беріліп, ол «True» (шын) константасымен анықталады. Басқа арнайы атом NIL, бір жағынан, «FALSE» (өтірік) константасын анықтаса, ал екінші жағынан, бос тізімді анықтайды. Құрамды өрнектер бұтақталған құрылымдарда беріледі. Деректерді сипаттайтын құрылым ретінде тізімдердің жақсы жағына мыналарды жатқызамыз: олар әртүрлі типтегі элементтерді біріктіреді, кез келген өлшемде (ұзындығы, көлемі) бола алады. Мысалы, Лисп-те мынандай тізім болуы мүмкін: («a»(9) ( ) N (? (WOMBAT)) `(A.B) NIL 0.9). Бұл тізімдегі элементтер құрамы әртүрлі: жолдар, әртүрлі форматтағы сандар, атомдар бульдық мәндер, т.б. болады.

Дегенмен тізімдерді пайдаланудың кейбір кемшіліктері де бар. Лисп-те олар стек түрінде болады да, тізімге ену тек біржақты түрде өтеді. Тізімдегі элементтерде массив элементеріндегідей қатынау мүмкіндігі болмайды. Яғни деректерді іздеп табуға қиындық туатындықтан, Лисп-те көлемі үлкен жиынды деректерді беру үшін, басқа да құрылымдар енгізілген. Қазіргі кездегі Лисп нұсқаларында массивтер, ХЭШ-кестелер, жазба тәрізді құрылымдар бар [14].

Лисп тілінің негіздері. *Лисп тіліндегі рекурсия.* Таңбалы өрнектер – Лисп тілінің синтаксистік негізі. Лисп программалау тілінің синтаксистік элементтеріне мыналар жатады: таңбалы өрнектер (symbolic expression), оларды кейде s-өрнектер (s-expression) деп атайды. Осы s-өрнектер түрінде программа да, деректер де жазылады, s-өрнектер түрінде атом (atom) немесе тізім (list) бейнеленеді. Лисп-тегі атомдар – құрамында сан және таңба бар тілдің базалық синтаксистік бірліктері. Таңбалық атомдар әріптен, цифрдан және әріптік-цифрлық емес таңбалардан * - + / 9 $ % « & <> – тұрады [11].

Лисп атомдарының мысалдарын келтірейік:

LISP. 3.1416
100
x
hyphenated-пате *some-global* nil

Лисп тіліндегі тізім дегеніміз – бір-бірінен бос орын арқылы бөлініп жақшаларға алынған атомдар тізбегі немесе басқа тізімдер. Мысалы:

(1 2 3 4)

(tom mary john joyce) (a (b c) (d (e E))) ( ).

Тізім элементтері есебінде басқа тізімдер де болуы мүмкін. Бұл жағдайда тізімдердің бір-біріне ену тереңдігі кез келген болады. Бұл жағдай күрделілігі әртүрлі кез келген қалыптағы таңбалық құрылымдар жасауға мүмкіндік береді. Деректер құрылымдарын құруда және оларды басқаруда бос тізімнің ( ) атқаратын рөлі ерекше. Бұл тізімнің өзінің арнайы аты болады: ***-nil***; Көрсетілген ***nil*** – бұл бір уақытта атом да, тізім де бола алатын s-өрнектер. Тізім бейнелеу құрылымдарын жасауға арналған өте икемді құрал. Мысалы, оны предикаттар теориясындағы өрнектерді бейнелеуге пайдалануға болады.

(on block-1 table)

(likes bill X)

(and (likes george kate) (likes bill merry)).

Бұл синтаксис предикаттар теориясындағы өрнектерді бейнелеуге ыңғайлы. Дерекқор колданбаларына қажет деректер құрылымын іске асыратын тізімдерді пайдалану тәсілдерін көрсететін екі түрлі мысал қарастырайық.

((2467 (lovelace ada) programmer) (3592 (babbage charles) computer-designer))

((key-1 value-1) (key-2 value-2) (key-3 value-3))

Лисп тілінің маңызды қасиетіне деректерді де, программаны да бейнелеуге бірдей синтаксисті пайдалану жатады. Мысалы, мына тізімдерді:

(* 7 9)

(- (+ 3 4) 7)

префистік қалыптағы жазба түріндегі арифметикалық өрнектер деп атауға болады. Лисп бұл өрнектерді былайша өңдейді: (*7 9) 7-нің 9-көбейтіндісі. Егер компьютерде Лисп орнатылса, онда пайдаланушы оның интерпретаторымен интербелсенді диалог (сұхбат) жүргізеді. Интерпретатор шақыру таңбасын береді – « > ». пайдаланушы енгізген деректі оқиды, оны бағалайды және нәтижені береді. Мысалы:

```
>(*79)
63
>
```

Мұнда пайдаланушы (*7 9) өрнегін енгізді, интерпретатор нәтижені 63 берді. 63 – осы өрнектің мәні. Бұдан соң Лисп шақыру белгісін

көрсетіп, пайдаланушының не енгізетінін күтеді. Бұндай цикл *оқу – бағалау – шығару* (read-eval-print loop) циклі деп аталады және ол Лисп интерпретаторының негізін құрайды.

Тізімді алған соң Лисп интерпретаторы оның бірінші элементін функция аты, ал басқа элементтерін функция аргументтері есебінде талдауға тырысады. Мысалы, мына s-өрнегі *(f x y)*, әдеттегі мынандай математикалық жазба түрінде сәйкес келеді *f(x,y)*. Лисп тілінде шыққан мәні – осы функцияның өзінің аргументтері көмегімен тапқан нәтижесі болып табылады. Лисп тілінің саналы түрде бағаланатын өрнектері форма (form) деп аталады. Егер пайдаланушы Лисп тілінде бағалана алмайтын өрнекті енгізсе, онда экранға қате жөнінде хабар шығып, пайдаланушыға трассировканы орындап, түзетулер енгізуге мүмкіндік болады [12]. Лисп интерпретаторы болатын сеансының мысалы келесі түрде болады:

```
> (+14 5)
19
>(+1234)
10
> (-(+ 3 4) 7)
0
>(* (+25) (– 7 (/6 2)))
28
>(= (+23) 5)
t
> (> (* 5 6) (+ 4 5))
t
> (a b c)
Error: invalid function: a
```

Бұл келтірілген мысалдарда аргументтің өзі тізім бола алады, мысалы, (– (+ 3 4) 7). Бұл өрнек функциялар композициясын береді және былайша оқылады: «3 пен 4 қосу нәтижесінен 7-ні алу». «Нәтижесі» сөзінің ерекшеленуі мынаны білдіреді: функцияға аргумент есебінде (+ 3 4) s-өрнегі емес, оны бағалау нәтижесі беріледі.

Функцияны бағалағанда Лисп тілі алдымен оның аргументтерін бағалайды, сосын өрнектің 1-ші элементі арқылы берілген функцияны қолданады. Егер аргументтердің өздері функция болса, онда Лисп рекурсивті түрде осы ережені оны бағалау үшін қолданады. Осылайша Лисп тілі бір-біріне ену тереңдігі бойынша кез келген функцияларды шақыруды ұйымдастыра алады.

Лисп тілінде мынаны еске ұстаудың маңызы бар: Лисп тілі үнсіз келісім бойынша барлық нысандарды бағалай алады. Бұл кезде мынандай келісім сақталады: сандар өз мәндеріне сәйкес келеді. Мысалы, егер Лисп интерпретаторына 5 мәнін берсек, онда Лисп нәтижеге де 5 мәнін қайтарады. Таңбалар мен мысалы X өрнектер байланысты болуы мүмкін.

Байланысқан өрнектерді бағалағанда байлану нәтижесі қайтарылады. Байланған таңбалар функцияны шақыру нәтижесінде пайда болады. Егер таңба байланыста жоқ болса, онда оны бағалауды қажет екені туралы хабар шығады.

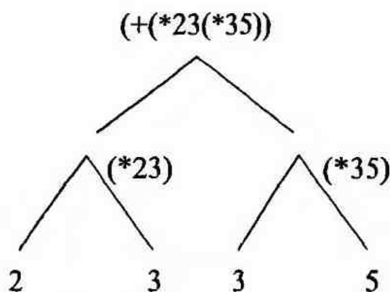
Мысалы, (+ (*2 3) (*3 5)) өрнегін бағалағанда Лисп алдымен (*2 3) және (*3 5) аргументтерін бағалайды. 1-ші аргумент үшін Лисп 2 және 3 параметрлерін бағалайды, яғни арифметикалық мәнін қайтарады. (*2 3) екеуі көбейтіліп, нәтижесінде 6 санын, (*3 5) көбейтіліп, нәтижесінде 15 санын береді. Бұл екі нәтиже жоғарғы деңгейде орналасқан қосу функциясына аргумент есебінде беріліп, нәтижесінде 21 қайтарылады. Осы операциялардың орындалуы 2.1-суретте көрсетілген.

Сонымен, осы тақырыптың негізгі ойларын келесі анықтамаларда тұжырымдауға болады [13].

S-өрнектерінің анықтамасы.

S-өрнегін рекурсивті түрде былайша анықтауға болады.

1) Атом – бұл S-өрнек.
2) Егер s1, s2 ... - S-өрнек болса, онда S-өрнегі тізім де бола алады.



2.1-сурет. Лисп қарапайым функциясын бағалайтын бұтақ диаграммасы

Тізім – атомдық емес S-өрнек. Форма дегеніміз – бағалауға тиісті S-өрнек. Егер бұл тізім болса, онда оның 1-ші элементі функция аты есебінде, ал одан кейінгі элементтер аргументтері есебінде жүреді.

Өрнектерді бағалау былайша орындалады: Егер S-өрнегі сан болса, онда осы сан мәні қайтарылады. Егер S-өрнегі атомдық таңба болса, онда онымен байланысқан мән қайтарылады. Егер таңба байланыста болмаса, қате туралы хабар шығады. Егер S-өрнегі тізім болса, оның 1-ші аргументінен басқасы бағаланады, алынған нәтижеге 1-ші аргумент анықтайтын функция қолданылады. Лисп тілінде S-өрнегі түрінде программа да, деректер де беріледі. Бұл қасиет тіл синтаксисін оңайлатады. Яғни деректер есебінде басқа программаларды өңдейтін программалар жазуда көп көмегін тигізеді. Бұндай программаларды S-өрнекті бағалауды басқару арқылы жазуға болады. Сонымен, осы аталған қасиет Лисп интерпретаторын іске асыруды жеңілдетеді.

Функциялар, тізімдер және таңбалық есептеулер. Осы уақытқа дейін Лисп тілінің синтаксисі мен бірнеше пайдалы функцияларды қарап өттік. Олардың барлығы қарапайым арифметикалық есептерді шешуге пайдаланылады. Дегенмен Лисп тілінің нағыз күші таңбалы есептеулерде жатыр. Бұл есептеулер таңбалық және сандық атомдардан тұратын кез келген күрделіліктегі құрылымдар құруға және оларды басқаратын формалар құруға арналған тізімдерді пайдалану негізінде жасалған. Таңбалы деректер құрылымын өңдеуді және Лисп тіліндегі абстракты деректер типін құруды қарапайым мысал арқылы қарастырып өтелік. Мысалы: деректер қорын басқару [13]. Қызметкерлер жайында мәліметтер бар жазбаларына операциялар орындалады. Өрістер жазбаларында қызметкердің аты-жөні, жалақысы және табельдік нөмірі берілген. Бұл жазбалар тізім түрінде бейнеленген. Тізімнің бірінші үш элементі: аты-жөні, жалақысы және табельдік нөмірі.

Nth функциясы көмегімен әртүрлі өрістер жазбаларына қатынауға болады. Мысалы, аты-жөні өрісін алу функциясын бейнелейік:

```
(defun name-field (record)
(nth 0 record))
```

Оны пайдалану мысалы мынандай:

```
(name-field '((Aigan Aidos) 45000.00 338519))
(Aigan Aidos).
```

Осыған ұқсас жалақы salary-field және табельдік m нөмер өрістеріне қатынау number-field функцияларын жазуға болады.

Аты-жөні деп аталатын өрістің өзі екі элементтен тұратын тізім түрінде беріле алады. Мысалы: (Аты, фамилиясы).

Енді name параметрі арқылы не атын, не фамилиясын анықтайтын функцияны енгізейік.

```
(defun first-name (name)
(nth 0 name))
```

Оны пайдалану мысалы мынандай:

```
> (first-name (name-field '((Aigan Aidos) 45000.00 338519)))
Aigan.
```

Сонымен, бірге жазбаларды құратын және жөндейтін функциясын да анықтау керек [12]. Бұл функциялар Лисп тілінің ішкі функциясы – list көмегімен анықталады. List функциясы кез келген мөлшерлі аргументтер санымен жұмыс істейді. Ол алынған аргументтерді бағалайды, сосын параметрлер мәнін өз элементі есебі ретінде жинақтаған тізіммен қайтарып береді. Мысалы:

```
> (list 1 2 3 4)
(1234)
> (list '(Aigan Aidos) 45000.00 338519)
((Aigan Aidos) 45000.00 338519)
```

Осы List функциясын дерек кордағы жазбалар конструкторын анықтау үшін пайдалануға болады.

```
(defun build-record (name salary emp-number)
(list name salary emp-number))
```

Оны пайдалану мысалы мынандай:

```
> (build-record '(Alan Turing) 50000.00 135772)
((Alan Turing) 50000.00 135772).
```

Енді қатынау функциясы және build-record көмегімен өзгертілген жазба көшірмесін қайтаратын функцияны жаза аламыз.

```
(defun replace-salary-field (record new-salary)
(build-record (name-field record)
new-salary
(number-field record)))
> (replace-salary-field '((Aigan Aidos) 45000.00 338519) 50000.00)
((Aigan Aidos) 50000.00 338519).
```

Бұл функция жазбаның өзін жөндемейді, оның жөнделген көшірмесін жасайды. Осы жөнделген көшірмені орасан үлкен айнымалымен байланыстыру арқылы сақтауға болады. Бұл байланысты set f функциясы атқарады. Лисп тілінде бастапқы құрылымдағы тізімнің нақты элементін жөндеуге (көшірме жасамай) мүмкіндік беретін формалар да бар. Дегенмен программалаудың жақсы стилінде оларды пайдалану қарастырылмайды. Егер қолданбаларда көлемі аса үлкен емес құрылымдар қолданылса, оларды жөндеу осы құрылымның көшірмесін жасау арқылы орындалады. Жоғарыда көрсетілген қызметкерлер жа-

йында мәліметтерді өңдейтін мысалда деректердің абстракты түрі құрылды. Деректер құрылымы элементтеріне қатынайтын және оларды жөндейтін әртүрлі функциялар құрылды. Программалаушы адам деректер қорындағы жазбаларды іске асыруды пайдаланатын тізімдердің нақты құрылымы жайында басын қатырмайды. Бұл жоғары деңгейдегі кода құруды жеңілдетеді және бұл коданы түсінуге және жөндеуге мүмкіндік береді.

Жасанды зерде қолданбалары сала аумағына қатысты үлкен көлемді білімдерді өңдеуді қажет етеді. Бұл білімдерді бейнелеуге қажет деректер құрылымдарын, мысалы, семантикалық торларды сипаттау белгілі бір қиындықтар тудырады. Әдетте, адам баласына білімдерді оның ішкі бейнелеуінің нақты синтаксисі бойынша емес білімдерді мағыналық терминдер арқылы өңдеу оңтайлы. Сондықтан жасанды зерде қолданбаларын құрғанда, деректердің абстракты типін құру әдіснамасының (методология) атқаратын рөлі өте маңызды. Лисп тілі жаңа функцияларды жеңіл түрде анықтай алатындықтан, бұл тіл деректердің абстракты типін құруға арналған ең тиімді тіл бола алады [13].

Тізімдер және рекурсивті құрылымдар. Жоғарыдағы келтірілген қызметкерлер жайлы мәліметтері бар мысалында қарапайым дерек өрістерінің жазбаларына қатынауды іске асыратын nth және list функцияларын пайдаландық. Қызметкерлер туралы жазбалардың барлығының тұрақты ұзындығы болғандықтан (үш элементтен тұрады), осы аталған екі функцияны қарастыру жеткілікті болады. Бірақ ұзындығы тұрақсыз болатын жазбалар үшін, бұл функциялар аз. Бұндай операцияны орындау үшін, тізімді итеративті немесе рекурсивті түрде сканерлеу мүмкіндігі болуы қажет. Яғни белгілі бір шарт орындалғаннан кейін (қажет жазба жазылған соң) немесе тізімді толық қарап шыққан соң операциялар орындалуы керек. Енді тізімдерді өңдеуде рекурсия қолданылатын функцияларды анықтайтын тізімдермен жұмыс істейтін операцияларды қарастырайық. Тізім бөліктеріне қатынауды іске асыру функцияларына car және cdr жатады. 1-ші бір аргументке тәуелді және өзі де тізім, 1-ші элементті қайтарады. Cdr – функциясы да бір аргументке тәуелді, ал тізімнің бірінші элементін алып тастағаннан кейінгі тізімді қайтарады [11]. Мысалы:

```
> (car '(a b c)) ; тізім квотталған
a
> (cdr '(a b c)) (b c)
> (car '((a b) (c d))) ; тізімнің 1-ші элементі тізім болуы мүмкін
```

```
(a b)
> (cdr ((a b) (c d)))
((c d))
> (car (cdr '(a b c d)))
b
```

car және cdr функциялары тізімдік құрылымдармен рекурсивті түрде жұмыс істеуге негізделген. Ондағы тізімнің әр элементіне арналған операциялар келесі түрде орындалады.

1) Егер тізім бос болса, операция аяқталады.

2) Қарсы жағдайда тізім элементі өңделіп, қадам жолы тізімнің келесі элементіне өтеді.

Осы сызбанұсқа бойынша тізіммен жұмыс істейтін көптеген функцияларды анықтауға болады. Мысалы, Common LISP(CL) тілі кейбір S-өрнегінің тізімге жатуын анықтайтын member предикатын, тізім ұзындығын табатын length предикатын өз құрамында ұстайды. Осы функциялардың басқа нұсқаларын қарап өтелік, яғни ол my-member функциясы екі аргументтен тұрады, кез келген S-өрнегінен және my-list тізімінен, егер S-өрнегі my-list тізімінің элементі болмаса, онда ол nil-қайтаруы керек. Қарсы жағдайда функция S-өрнегін 1-ші элемент есебінде ала отырып, тізімді айналдыра береді.

```
defun my-member (element my-list)
(cond ((null my-list) nil); элемент тізім мүшесі бола алмайды
((equal element (car my-list)) my-list); элемент табылды
(t (my-member element (cdr my-list))))); рекурсия қадамы
Мысалы, my-member функциясын пайдаланып көрейік.
(my-member 4 '(1 2 3 4 5 6)) 5 6)
(my-member 5 ' (a b c d)).
```

Осыған ұқсас length және nth функцияларының нұсқаларын анықтаймыз.

```
defun my-length (my-list)
(cond ((null my-list) 0)
(t (+ (my-length (cdr my-list)) 1))))
defun my-nth (n my-list)
(cond ((zerop n) (car my-list)) ; zerop функциясы тексереді
;барлық аргументтің 0-ге тең екенін
(t (my-nth (-n 1) (cdr my-list))))).
```

Жоғарыда келтірілген мысалдардағы car және cdr функциясын пайдалану Лисп тілінің даму тарихын да баяндай алады. Тілдің ең алғашқы нұсқаларында Common LISP(CL) тіліндегідей көптеген ішкі

функциялар болған жоқ. Элементтің тізімге жатуын анықтау, тізім ұзындығын есептеу т.б.с.сияқты функцияларды программалаушы өзі құрып отырады. Уақыт өте келе, осындай көптеген функциялар тіл стандартына енді. Common LISP(CL)тілі – оңай кеңейетін тіл. Ол қайта пайдаланылатын функциялардың өз кітапханасын құруға және оны пайдалануға мүмкіндік береді [11].

Car және cdr-ден басқа Лисп тілінде тізімдерді құратын функциялар жиынтығы бар. Соның бірі – list. Ол кез келген мөлшердегі аргументтері бар өрнек болып табылады. Функция оларды бағалайды да, нәтижелер тізімін қайтарады. Тізімдердің қарапайым конструкторы болып cons-функциясы табылады. Бұл функцияның параметрлері ретінде екі S-өрнегі жүреді. Бұл параметрлер бағаланады да, нәтиже ретінде тізім қайтарылады. Тізімнің 1-ші элементі ретінде 1-ші аргумент мәні, ал тізім құйрығы ретінде 2-ші аргумент мәні жүреді.

```
> (cons 1 '(2 3 4))
~ (2 3 4)
> (cons '(a b) '(c d e))
((a b) c d e)
```

Cons-функциясын car және cdr функциясының кері түрлендіруі деп есептеуге болады, өйткені car функциясын cdr функциясының қайтару мәніне қолдану нәтижесінде cons-функциясының 1-ші аргументін аламыз. Осыған ұқсас cdr функциясын cons қайтару формасына қолдану нәтижесінде осы форманың 2-ші аргументін аламыз.

```
> (car (cons 1 '(2 3 4)))
1
> (cdr (cons 1 '(2 3 4)))
(2 3 4)
```

Cons-функциясын пайдалану мысалын қарастырайық. Ол үшін filter-negatives функциясын анықтайық. Оның параметрі есебінде сандық тізім жүреді. Бұл функция тізімдегі барлық теріс мәндерді жояды. Filter-negatives функциясы тізім элементтерін рекурсивті түрде тексереді. Егер 1-ші элемент теріс болса, онда ол алынып тасталады да, функция тізім құйрығының теріс элементтерінің сұрыптау нәтижесін қайтарады. Тізім құйрығы cdr функциясы арқылы анықталады. Егер тізімнің 1-ші элементі оң болса, онда ол сұрыпталған құйрықтағы теріс сандар нәтижесіне қосылады.

```
((defun filter-negatives (number-list)
(cond ((null number-list) nil) ; тоқтау шарты
((plusp (car number -list)) (cons (car number-list)
```

```
(filter-negatives (cdr number-list))))
(t (filter-negatives (cdr number-list)))))
```

Осы функцияның шақырылу мысалы:

```
> (filter-negatives '(1 -1 2 -2 3 -4))
(1 2 3).
```

Бұл мысалда cons-функциясын тізімдердегі рекурсивті функцияларға қолданылуы көрсетілген. Car және cdr функциялары тізімді бөліктерге бөледі де, рекурсияны «басқарады», ал cons-рекурсия ары қарай өрістегенде нәтижені қалыптастырып отырады. Cons-функциясын пайдаланудың тағы бір мысалы – ішкі append функциясын қайтадан анықтау.

```
(defun my-append (listl list2)
(cond ((null listl) list2)
(t (cons (car listl) (my-append (cdr listl) list2)))))
```

Осы функциялардың шақыруын көрсетейік.

```
> (my-append ' (1 2 3) ' (4 5 6) )
(1 2 3 4 5 6)
```

My-append, my-length, my-member функциясын анықтау осы жоғарыда аталған рекурсивті сызбанұсқаға ұқсас жүреді. Әр анықтауда тізімдегі элементті алып тастау үшін, cdr функциясы қолданылады. Бұл қысқартылған тізімді рекурсивті түрде шақыру мүмкіндігін қамтамасыз етеді. Егер тізім бос болса, рекурсия аяқталады. Рекурсия үдерісі кезінде cons функциясы шешіледі де, қайтадан қарап отырады. Бұл сызбанұсқаны әдетте cdr рекурсиясы (cdr-recursion) деп немесе құйрықтық рекурсия (tail recursion) деп атайды. Өйткені cdr функциясы тізім құйрығын алу мүмкіндігін береді.

2.8. Лисп тіліндегі есептеулер мен функционалдар

Лисп тіліндегі бағалау: quote және eval функциялары. Әдетте, аргументтер тізімінің алдында бір тырнақша таңбасы «'» қойылады. Бұл таңба guote функциясын бейнелейді, яғни бұл – өз аргументтерінің бағалауын болдырмайтын арнайы функция. Бұл аргументтер бағаланатын форма емес, деректер түрінде өңделуге тиіс екенін білдіретін арнайы таңба. S-өрнегін бағалай отырып, Лисп тілі алдымен барлық аргументтерді бағалауға тырысады. Егер интерпретаторға (nth 0 (a b c d)) өрнегі берілсе, онда ол алдымен (a b c d) аргументін бағалауға тырысады. Бұл қатеге әкеледі, өйткені a – S-өрнегінің 1-ші элементі Лисп функциясы емес. Осының алдын алу үшін guote ішкі

функциясын қолданады. Ол бір аргументке тәуелді және осы аргументі бағаламай, сол күйінде кайтарады [12]. Мысалы:

```
> (quote (a b c))
(a b c)
> (quote (+ 1 3))
(+ 1 3)
```

Quote функциясы өте жиі қолданылатындықтан, оның орнына «'» – таңбасы пайдаланылады. Сондықтан жоғарыдағы мысалдарды былайша жазуға болады:

```
> '(a b c)
(a b c)
> '(+ 1 3)
(+ 1 3).
```

Жалпы guote функциясы аргументтерді бағалауды болдырмауға қызмет етеді және аргументтер форма емес, деректер түрінде өңделуі керек. Алдыңғы қарастырған мысалдардағы қарапайым арифметикалық амалдар үшін guote функциясы қажет емес, өйткені сандар әр кезде өз мәнімен бағаланады. Quote функциясын пайдаланудың list функциясын шақырудағы мысалдарын қарап өтейік.

```
> (list (+ 1 2) (+ 3 4))
(3 7)
> (list '(+ 1 2) '(+ 3 4))
((+ 1 2) (+ 3 4))
```

Көрсетілген 1-мысалда аргументтер квоталанбаған, сондықтан олар list функциясына үнсіз келісім бойынша қолданылатын сызбанұсқа бойынша беріледі. 2-ші мысалда guote функциясы бағалауды болдырмайды да, list функциясының аргументтері есебінде S-өрнегі беріледі. (+1 2) Лисп тілінің формасы болса да, guote функциясы оның бағалауын болдырмайды. Программаны бағалауды болдырмау және оларды деректер есебінде қарау – Лисп тілінің ең маңызды қасиеті. Quote функциясының қосымшасы ретінде Лисп тілінің S-өрнегі бағалайтын eval функциясын береді. Ол бір аргументке тәуелді, ол S-өрнегі болады. Бұл аргумент функцияның әдеттегі аргументі есебінде бағаланады. Дегенмен алынған нәтиже тағы бағаланады да, eval өрнегінің мәні ретінде ең соңғы нәтиже анықталады. Eval және guote функциясының мысалдарын келтірейік:

```
> (quote (+ 2 3) )
(+ 2 3)
> (eval (guote (+ 2 3)))      ; eval функциясы нәтижені алып тастайды
5                             ; guote функциялары
```

```
> (list '(* 2 5)        ; бағаланған S-өрнегі құрылады
(* 2 5)
> (eval (list '* 2 5)) ; бағаланған өрнек құрылады
10
```

Eval функциясы S-өрнегінің әдеттегі бағалауын жүргізеді. Eval және guote функциялары арқасында метаинтерпретаторларды (meta-interpreter) жасау оңайланады. Бұл Лисп тілі стандартты интерпретаторының вариациялары бола алады. Олардың көмегімен программалаудың осы тілінде қосымша және балама мүмкіндіктерді құруға болады.

Лисп тілінде программалау: жаңа функциялар құру. Common LISP диалектісінде көптеген ішкі функциялар бар [11]. Оның ішінде:

1) бүтін санды, нақты санды және комплексті сандар арифметикасына арналған арифметикалық функциялардың толық спектрі;
2) программаны басқаратын және цикл ұйымдастыратын әртүрлі функциялар;
3) тізімдер және баска да құрылымдармен жұмыс істейтін функциялар
4) енгізу-шығару функциясы;
5) функцияны бағалауды бақылауға арналған форма;
6) операциялық жүйе және ортаны басқару функциясы.

Лисп тілінің барлық функциялары туралы мәлімет арнайы әдебиеттерде келтіріледі [11, 12, 13, 14].

Лисп тілінде программалауда жаңа функциялар анықталады және программа құруда пайдаланушы және ішкі функциялар қолданылады. Жаңа функциялар defun функциясы көмегімен анықталады. Бұл «define function» деген сөзді білдіреді (яғни функцияны анықтау). Функцияны анықтап болған соң, оны тілдің баска ішкі функциялары сияқты пайдалануға болады. Мысалы, бір аргументке тәуелді, осы аргументтің квадратын алатын square функциясын анықтау қажет болсын дейік. Лисп тілінде ол былайша анықталады:

```
(defun square (x)
(* x x))
```

Defun функциясының 1-ші аргументі анықтайтын функцияны атап береді, 2-ші аргумент оның формальды параметрлерінің тізімін береді. Олар таңбалы атомдар болады. Қалған аргументтер – нөл немесе жаңа функцияның денесін құратын S-өрнегі, яғни оның тәртібін анықтайтын Лисп кодасы. Лисп тілінің басқа функциялары сияқты defun функциясы жаңа функция атын беретін нәтижені қайтарады, defun функциясының жұмыс істеуінің маңызды нәтижесіне оның жаңа функцияны құрып,

оны Лисп тілі ортасына қосуы жатады. Жоғарыда көрсетілген мысалда бір аргументке тәуелді және осы аргументтің өзіне өзін көбейту нәтижесін қайтаратын square функциясы анықталған. Функцияны анықтап болған соң, оны сол аргументтер санымен немесе фактідегі параметрлер санымен шақыру керек. Функцияны шақырғанда фактідегі параметрлер формальді параметрлерімен байланысады да, функция денесі осы байланыс нәтижесінде бағаланады.

Мысалы, (square 5) шақырылуында 5 мәні анықталу денесіндегі X-пен байланысады. (* x x) денесін бағалай отырып, Лисп тілі функция аргументтерін бағалайды. Бұл шақыруда X-параметрлері 5 мәнімен байланысады да, (* 5 5) өрнегі бағаланады.

Жалпы Defun өрнегінің қатаң синтаксисі былай бейнеленеді:

```
(defun <функция аты> (<формальды параметрлер>)
<функция денесі>).
```

Бұл анықтамада форма элементінің сипаты < > – жақшасының ішінде. Осы белгілеу Лисп тілінде үнемі пайдаланылады. Defun – функциясының формальды параметрлері тізім түрінде беріледі [12].

Жаңадан анықталған функцияны басқа да ішкі функциялар сияқты пайдалануға болады. Мысал қарастырайық: тік үшбұрыштың екі жағының ұзындығын біле отырып, гипетенуза ұзындығын есептеп шығару керек. Бұл функцияны Пифагор теоремасы негізінде анықтауға болады. Ол үшін біз анықтаған square функциясын пайдаланамыз. Келесі көрсетілген Лисп тілі кодасы түсіндірмемен берілген. Түсіндірме « ; » таңбасынан кейін басталады.

```
(defun hypotenuse (x y)    ; гипотенуза ұзындығы тең
(sqrt (+ (square x)        ; қосындыдан алынған квадрат түбір
(square y))))              ; басқа екі жағы.
```

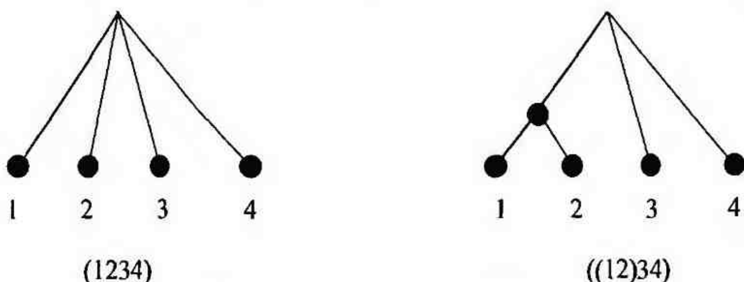
Бұл көрсетілген мысал Лисп тілі үшін типтік бола алады, өйткені Лисп тілінде программалар жақсы анықталған есептерді шешетін кішкене функциядан тұрады. Осы функция анықталған соң, одан жоғары деңгейдегі функциялар анықталады т.б.с.с.

Енгізілген тізімдер, құрылымдар және рекурсия car-cdr. Мына екі функция: cons және append кішкене тізімдерді біріктіруге арналғанымен, екеуінің өзгешеліктері бар. Cons функциясын шақырғанда оның параметрлері екі тізім болады. 1-шісі 2-ші тізімнің 1-ші элементі болады. Ал append функциясы тізімді айналдырады. Тізім мүшелері ретінде екі аргумент элементтері жүреді, [13]. Мысалы:

```
> (cons ‘ (1 2) ‘ (3 4))
((1 2) 3 4)
```

```
> (append '(1 2) '(3 4))
(1 2 3 4)
```

Көрсетілген (1 2 3 4) және ((1 2) 3 4) тізімдерінің құрылымдары принциптік түрде әртүрлі. Бұл өзгешелікті граф түрінде бейнелеуге болады. Ол үшін тізімдер мен ағаштар арасындағы сәйкестікті орнатса жеткілікті. Ол үшін әр тізім үшін түйін жасау керек. Бұл түйін туындылары тізім элементтері болады. Бұл ереже рекурсивті түрде барлық тізімдер элементтеріне қатысты, яғни элементтердің өздері тізім болса да, осылайша тізімді құрсақ, жоғарыда келтірілген екі тізім екі түрлі ағашты құрады (2.2-суретті қара).



2.2-сурет. Құрылымы екі түрлі тізімдердің ағаш түрінде бейнеленуі

Бұл мысал деректерді бейнелеудегі тізімдер мүмкіндіктерінің зор екендігін білдіреді. Мысалы, бір-біріне енгізілген тізімдер – күрделі деректердің иерархиялық құрылымын сипаттауда таптырмайтын құрал. Атап айтсақ, қызметкер жайлы мәліметтер мысалындағы «Аты» деген өрістің өзі тізім болады, ол «Аты» «Фамилиясы» деген екі өрістен тұрады. Бұл тізімді бір ұғым деп қабылдап, оның әр бөлігіне қатынап тұруға болады.

Ал, cdr-рекурсиясының қарапайым сызбанұсқасы бір-біріне енгізілген тізімдердің барлық операцияларына жарай бермейді. Мысалы, length функциясын алайық. Оны мына иерархиялық құрылымға қолданайық:

```
> (length '((1 2) 3 (1 (4 (5)))))
3
```

Осы length функциясы 3 мәнін қайтарады, өйткені тізім 3 элементтен тұрады: (1, 2), 3 және (1 (4 (5))). Басқа жағынан егер тізімдегі атомдар санын білу қажет болса, онда рекурсияның басқа тізімін анықтау қажет. Бұл сызбанұсқада тізім элементтерін анықтағанда тізімнің атомдық емес элементтері ашылады және атомдар санын есептейтін операциялар орындалады. Бұл функцияны былай анықтауға болады:

```
defun count-atoms (my-list)
(cond ((null my-list) 0)
((atom my-list) 1)
(t (+ (count-atoms (car «пу-list))     ;элементті ашамыз
(count-atoms (c dr my-list))))))       ;тізімді сканерлейміз
> (count-atoms '((1 2) 3 (((4 5 (6)))))
6
```

Осы анықтама car-cdr-рекурсиясының мысалы бола алады. Cdr функциясы көмегімен тізімде рекурсивті түрде өтулер болады және count-atoms функциясы 1-ші элемент бойынша рекурсияны орындайды. Ол car формасы бойынша алынған болатын. Содан соң «+» – функциясы жауапты қалыптастырғанда екі нәтижені біріктіреді. Рекурсия атомға жеткенде немесе тізім соңы болғанда аяқталады. Бұндай сызбанұсқаны тізімнің әрбір элементіне «тереңдеуді» қамтамасыз ететін қарапайым cdr рекурсиясына екінші өлшемді қосу деп те қарастыруға болады [14].

Енді car-cdr-рекурсиясын пайдаланатын тағы бір мысал ретінде flatten функциясын анықтауды қарастыруға болады. Ол бір аргументке тәуелді, әртүрлі құрылымдағы тізім иерархиялық емес тізімді қайтарады, сол берілген тәртіппен орналасқан атомдардан тұрады.

Ал flatten және count-atoms функцияларының анықтау ұқсастығына көңіл аударыңыз. Екі функцияда да тізімдерді бөлуге және рекурсияны басқаруға car-cdr сызбанұсқасы қолданылады. Рекурсия нөлдік немесе атомдық элементіне жеткенде аяқталады. Жауапты құру үшін рекурсивті шақыру нәтижесінде екі жағдайда да екінші бір функция қолданылады (ол append немесе +).

```
(defun flatten (1st)
(cond ((null 1st) nil)
((atom lst) (list lst))
(t (append (flatten (car lst))(flatten (cdr 1st)))))).
```

Мысалы, flatten функциясының шақырылуын қарастырайық.

```
> (flatten '(a (b c) (((d) e f))))
(a b c d e f)
> (flatten '(a b c)) ;бұл тізімде иерархия жоқ
(ab c)
> (flatten ' (1 (2 3) (4 (5 6) 7) ) )
(1 2 3 4 5 6 7)
```

Айнымалыларды set функциясы көмегімен байлау. Лисп тілі рекурсивті функциялар теориясы негізінде құрылған. Оның ең алғашқы нұсқалары функциялардың немесе қолданбалық программалау тілі бола алады. Таза функционалдық тілдер аспектісіне функцияны орындау нәтижесінде жанама болатын әсерлер жоқтығы жатады. Ол – функция қайтаратын мән функция анықтауы мен оны шақырғандағы нақты параметрлерден тәуелді деген сөз. Лисп тілі математикалық функцияларға негізделсе де, оның бұл қасиетін бұзатын формаларды да анықтауға болады. Лисп тіліндегі келесі командаларды қарастырып өтелік:

```
> (f 4)
5
> (f 4)
6
> (f 4)
7
```

Мұндағы f-толық мәнді функция емес, өйткені оның әр шақырудағы мәні әртүрлі. Параметрі – 4. Өйткені бұл функция орындалғанда жанама эффект болады да, ол оның әр шақыруындағы тәртібіне әсер етеді. Бұл f функциясы Лисп тіліндегі set ішкі функциясы көмегімен анықталған.

```
(defun f (x)
(set 'inc (+ inc 1))
(+ x inc))
```

Бұндағы set функциясы аргументке тәуелді. 1-шінің бағалау нәтижесі таңба, ал 2-ші параметр кез келген S-өрнегі бола алады. set функциясы 2-ші аргументті бағалайды да, оған 1-ші аргумент анықтаған таңба мәнін қосады. Жоғарыда көрсетілген мысалды inc-айнымалысы мына шақырумен (set ' inc O) – іске асса, онда келесі шақыруда бұл параметр мәні 1-ге көбейіп отырады.

Ал set функциясын бағалау нәтижесінде 1-ші аргумент міндетті түрде таңба-символ болуы керек. 1-ші параметрі қарапайым квотталған таңба болады. Бұндай жағдай өте жиі кездесетіндіктен, Лисп тілінде setg функциясынын балама формасы болады, шақырылу кезінде 1-ші параметр бағаланбайды. Ал setg функциясын пайдалану үшін міндетті түрде оның 1-ші параметрі таңба-символ болуы керек. Мысалы, эквиваленттің келесі формасын қарастырайық.

```
> (set 'x 0)
O
```

```
> (setg x 0)
O
```

Лисп тілінде set функциясын пайдалану математикалық тұрғыдан таза болатын нысандарды құра алмаса да, оның айнымалы қоршауындағы мәндермен байланыса алу қасиеті өте пайдалы қызмет болып табылады. Программалаудың көптеген есептерінің күйін функцияларды шақыру аралығында сақтала алатын нысандарды анықтау нәтижесінде іске асырған дұрыс. Осындай нысандардың классикалық мысалы есебінде кездейсоқ сандар генераторлары тудыратын бастапқы сан бола алады. Оның мәні функцияны шақырған сайын өзгереді және сақталады. Осыған ұқсас деректер қорын басқару жүйесінде деректер қорының өзін орасан үлкен ортадағы айнымалымен байланыстыра отырып сақтау ыңғайлы [11].

Осылайша таңбаға мән берудің екі түрлі тәсілі бар: бірі – ашық түрдегі, яғни set және setg функциялары көмегімен, екіншісі – жабық тәсіл, яғни функцияны шақырғанда оның накты параметрлері функция анықтамасындағы формальды параметрлермен байланысады. Жоғарыда аталған мысалдарда функция денесіндегі барлық айнымалылар не байланған (bound variable) немесе бос (free variable) түрде анықталған. Байланған айнымалы деп функция анықтамасындағы формальды параметр есебінде қолданылатын, ал бос айнымалы деп функция денесінде кездесетін бірақ формальды параметр бола алмайтын айнымалыны айтамыз.

```
> (defun foo (x)
(inc x (+ x 1))        ; x байланған айнымалыны анықтау x қайтару
> (setg y 1)
1
> (foo y)
2
> y ; y мәні өзгермеген
1
```

Келтірілген мысалдардағы *f* функциясының *x* айнымалысы байланған, ал *inc* бос айнымалы болып табылады. Осы мысалда келтірілгендей, функция анықтамасындағы бос айнымалылар жанама әсерлер көзі бола алады. Балама есебінде *set* және *setq* формалары үшін мән беретін *set f* жалпылама функциясын алуға болады. Бұл функция таңбаға мән бермейді, жадыдағы орынды анықтау мақсатында бірінші аргументті бағалайды да, осы орынға екінші аргумент мәнін

қояды. Мәндер мен таңбаларды байланыстырғанда, *set f* функциясы өз қызметін setq функциясына ұқсас түрде орындайды.

```
> (setq x 0)
0
> (setf x 0)
0
```

Дегенмен кез келген форманың жадыдағы орнын сипаттау үшін *set f* функциясын шақыруға мүмкіндік болса да, ол одан да жалпы семантиканы орындай алады. Мысалы, егер *set f* функциясының бірінші параметрі есебінде car функциясын шақыру жүрсе, онда *set f* тізімнің бірінші элементін ауыстырады. Егер *set f* функциясының бірінші параметрі есебінде *cdr* функциясын шақыру жүрсе, онда *set f* тізімнің құйрығын ауыстырады. Мысалы, былай:

```
> (setf x '(a b c))        ; x тізіммен байланысады
(ab c)
> x                        ; x мәні – (a b c) тізімі
> (setf (car x) 1)         ; x үшін car функциясын шақыру жады
                             адресіне сәйкес келеді
> x                        ; setf функциясы бірінші x элементінің
                             мәнін өзгертті
(1 b c)
> (cdr x) ' (2 3))
> (setf (2 3)
> x                        ; енді x құйрығы өзгерді
(12 3)
```

Осы *set f* функциясын Лисп тілінің көптеген формалары үшін, егер ол жады адресіне сәйкес келсе, шақыруға болады. Бірінші параметр есебінде таңбалар мен функциялар жүре алады, мысалы, *car, cdr* және *nth* функциялары. Осылайша *set f* функциясы Лисп тілінде деректердің құрылымын құруда, оларды басқаруда және олардың бөліктерін өзгетуде үлкен икемділікті қамтамасыз етеді.

Жергілікті айнымалыларды let функциясы көмегімен анықтау. *let* функциясы – бұл айнымалыларды байлауды басқарудағы тағы бір пайдалы формалардың бірі. Ол жергілікті айнымалыларды құруға болысады. *let* функциясын пайдалану мысалы ретінде квадратты теңдеудің түбірін табу функциясын қарастырайық. *quad-roots* функциясы $ax + bx + c = 0$ теңдеуі үш параметрге тәуелді: *a, b* және *c*. Ол осы теңдеудің екі түбірінен тұратын тізімді қайтарады. Түбірлер мына формулалармен есептеледі:

```
> (quad-roots 1 2 1)
(-1,0 -1,0)
> (quad-roots 1 6 8)
(-2,0 -4,0)
```

Осылайша өрнектеу нәтижесінде quad-roots функциясының мәнін есептегенде, келесі өрнек: *y'6 – 4ac* екі рет қолданылады. Тиімді және жинақы түрде есептеу үшін, оның мәнін бір рет есептеп және оны жергілікті айнымалыда сақтап, екі түбірді есептеуде пайдалануға болады. Сондықтан *quad-roots* функциясының бастапқы іске асуы келесі түрде болады:

```
(defun quad-roots-1 (a b c)
(setq temp (sqrt ( — (* b b) (* 4 a c))))
(list (/ (+ (- b) temp) (* 2 a))
(/ (- (- b) temp) (* 2 a))))
```

Мынаны есте ұстайық: теңдеудің нақты емес (мнимый) түбірлері жоқ деп есептеледі. Теріс саннан түбір алу жағдайында sqrt функциясы қате кодасы бар хабар береді. Бұл жағдайда өңдеу басқа жолдармен жүргізіледі.

Осы қыстырманы ескере отырып, функция кодасын дұрыс деп есептегеннің өзінде функция денесін бағалау жанама әсер беруі мүмкін. Ол орасан үлкен қоршамадағы temp айнымалысына мән тағайындайды:

```
> (quad-roots-1 1 2 1) (-1,0 -1,0)
> temp 0.0
```

Одан көрі quad-roots функциясы үшін temp айнымалысын жергілікті етіп, жанама әсер түрін алып тастау пайдалы. Ол үшін let блогын қолдану қажет. Осы өрнектің синтаксисі келесі түрде болады:

(let (<жергілікті айнымалылар>) <өрнек>)

Тізім элементтері (*<жергілікті айнымалылар>*) – бұл таңбалық атомдар немесе (<таңба> *<өрнек>*) түріндегі жұптар.

Форманы немесе блокты бағалағанда, let жергілікті қоршаманы тағайындайды. Ол (*<жергілікті айнымалылар>*) тізімнің барлық таңбаларынан тұрады. Егер кейбір таңба жұптың бірінші элементі болса, онда екінші элемент бағаланады да, бағалау нәтижесі осы таңбамен байланысады. Жұпқа енбеген таңбалар nil мәнімен байланысады. Егер осы таңбалардың кейбіреуі орасан үлкен қоршамамен байланыста болса, онда бұл үлкен байланыстар сақталады және олар let блогынан шыққанда қайтадан қалпына келеді.

Осы жергілікті байланыстар орнатылғаннан кейін let формасының екінші параметрімен берілген *<өрнектер>* осы қоршамада берілген

тәртіп бойынша бағаланады. Let блогынан шыққанда, осы блокта бағаланған соңғы өрнектің мәні қайтарылады.

Енді let блогының жұмыс істеу тәртібін келесі мысалмен бейнелейік.

```
> (setq a 0)
0
> (let ((a 3) b)
> (setq b 4) (+ a b))
7
> a
0
> b
```

Бұл мысалда let блогы орындалғанға дейін айнымалы 0 мәнімен, ал қоршаманың жоғарғы деңгейінде *a b* мәндерімен байланысқан. Форманы бағалағанда let *a* – 3 мәнімен, ал *b* – *nil* мәнімен байланысады. Setq функциясы *b* айнымалысына 4 мәнін береді, содан соң let операторы *a* және *b* мәндерінің қосындысын қайтарады. Осы let формасы жұмысын аяқтаған соң *a* және *b* айнымалыларының алдыңғы мәндері қайтадан қалпына келеді, оның ішінде *b* айнымалысының байланбаған статусы да қалпына келеді. Let операторының көмегімен quad-roots функциясын жанама әсер тудырмай-ақ іске асыруға болады.

```
defun quad-roots-2 (a b c)
Let (temp)
(setq temp (sqrt (– (*b b) (* 4 a c))))
(list (/ (+ (– b) temp) (* 2 a))
(/ (– (– b) temp) (* 2 a))))
```

Жоғарыда көрсетілгендей, temp айнымалысын да let операторы арқылы байлауға болады. Ол quad-roots функциясының реттелген түрін іске асыруға мүмкіндік береді. Бұл жағдайда формуладағы бөлінгіш бір рет есептеледі де, оның мәні жергілікті denom айнымалысында сақталады.

```
defun quad-roots-3 (a b c)
Let ((temp (sqrt (– (*b b) (* 4 a c)))) (denom (* 2 a)))
(list (/ (+ (– b) temp) denom)
(/ (– (–b) temp) denom))))
```

Жанама әсерлерді алып тастаумен қоса, quad-roots-3 нұсқасы басқа іске асырылуларда да тиімді, өйткені онда біркелкі мәндерді қайталап есептеулер болмайды.

2.9. Функционалдық программалаудың қолданылуы

Функционалдық программалау неге қажет деген сұрақтар туған жағдайда оған қажетті жауаптарды программалаудың осы стиліндегі тілдердің бірінің жақсы қасиеттерін атап өтуден іздейік [16]. Функционалдық программалау тілдерінің бірі Haskell құрушылар оның көптеген жағымды жақтарын атап өтеді:

- бұл тілде күрделі программаларды жазу оңай және программалар қысқа болады;
- программаларды оқып тануға оңай; оларды Haskell тілін толық білмей-ақ түсінуге болады;
- қателер саны аз, өйткені Haskell тілінің синтаксисі программалаушы адамды көптеген әдеттегі қателерден сақтап отырады;
- программаларды құру және жобалау кезеңдері қысқа және қарапайым болады: ол үшін программалаушы оған қажеттіні анықтап, оны формальды математика тілінде сипаттауы қажет;
- адаптивті, оңай өзгертілетін және кеңейтілетін программалар құруға болады.

Сонымен бірге тілде қатаң типтеу болғандықтан, Haskell тілінің программаларында жүйелік қателер мен төтенше жағдайлар (core dump) болмайды деуге болады.

Haskell тілін құрушылардың айтуы бойынша бұл тілде жасалған программалар модульдерін басқа программаларға оңай кіріктіруге болады және оларды қайтадан пайдалану (*code reuse*) мүмкіндігі зор. Мысалы, Haskell тіліндегі тез сұрыптау программасы Си тіліне қарағанда бүтін сандарды ғана сұрыптап қоймай, салыстыру операциясы анықталған *Float* типіндегі сандарды да сұрыптайды [20].

Бұл тілдің тағы бір ерекшелігіне ондағы абстракция деңгейінің жоғары екенін атап өтуіміз қажет. Ол дегеніміз – функцияның өзін қайтаратын функцияларды құру мүмкіндігі. Тіпті дәлірек айтсақ, Haskell тілінде абстракты лямбда-есептеулер (л-есептеу) бар. Бұл есептеудің жойқын күші мен қуатын программа құрушылар әлі толығынан сезіп біле алмай жатыр.

Айта берсек функционалдық тіл ерекшеліктері көп-ақ. Бүгінгі таңдағы программалау тілдері индустриясының акценті белгілі бір қасиеттерге ауысты. Қазіргі кезде программаны орындау жылдамдығы немесе супер оңтайлы коданы жазу мүмкіндігі ешкімді таңғалдыра алмайды. Бүгінгі таңда алға қойылып отырған талаптардың бірі – программа құрушының өзі құрған кодасынан өзі шыға алмайтын

қиындықты жеңу немесе бір не бірнеше жыл өткен соң, өзінің не жазып не қойғанын білмей бас қатыруы емес, қайта осы көрсетілген кедергілерді жоя білу қабілеті жатады. Сондықтан программалау тілдеріне деген талаптардың ең маңыздысы және күрделісіне бұндай тілдердің табиғи тілге жақындығы жатады. Әдетте, іс жүзінде кейбір маңызды функциялардың қызметін арттырып, жылдамдату қажет болатыны да бар. Бірақ жалпы программалық кода мазмұнында олар көп орын алмайтындықтан, оларға аса көңіл бөлінбейтіні де анық. Мысалы, жоғарғы жылдамдық қажет болса, онда оны Си тілінде жазуға немесе кітапхана ретінде ұйымдастырып, негізгі қолданбаға кіріктіруге болады. Ал негізгі қолданбаны Python немесе Haskell тілінде (тез орындалуы үшін) жазуға болады.

Жадыны кіріктіру арқылы басқару. Қазіргі таңда көптеген программалау тілдерінде қоқысты жинаудың автоматты жүйелері бар. Яғни, басқаша айтқанда, жадыны автоматты басқару жүйесі бар. Көптеген күрделі программаларда жадыны динамикалық түрде бөліп беріп отыру қажеттігі туады. Бұндай бөлудің статикалық түрден ерекшелігі мынада: жадының қажетті көлемі программа орындалып тұрғанда анықталады. Содан соң операциялық жүйеге сұрату жасалып, қажетті байттар көлемі бөлінеді. Си тілінде мұндай жағдайда *malloc* функциясы қолданылады. Программа құрушы осы жады көлемін қайтадан жүйеге қайтуына жауап береді. Бөлінген жадыға қажеттілік болмағанда, программалаушы жадының босағаны туралы сұратуды жүйеге жібереді. Әдетте, осындай жадының бөлініп қайтарылатын бөліктері жайлы жіберілетін сұрату жауаптар мен сұрақтар программаның жұмыс істеу істемеу қабілетіне әсер етеді. Сонымен бірге жадыны бөлу программасын жасау «қымбат» дүниелерге жататындықтан, Си тілінде программалаушылар өздерінің жеке менеджері сияқты программа жазумен әуестеніп кетеді. Ол менеджер программалар әуелі жүйеден жадының көп көлемін алып, содан соң оны өз программасында бөлумен айналысады.

Ал функционалды тілдер программалаушыны осындай міндеттен құтқарады. Жады ашық емес автоматты түрде бөлінеді, ал арнайы қоқыс жинағыш (garbage collector) жүйеге пайдаланылмаған жады бөлігін қайтарады. Жадыны оңтайлы түрде басқару алгоритмдері күрделі болып келеді, бірақ қазір олар жақсы дамыған.

Си артықшылықтары. Әрине, функционалдық тілдердің де өз кемшіліктері бар. Функционалдық тілдерде жазылған программалар императивті тілдер прогаммаларына қарағанда баяулау жұмыс істейді

деген пікір қалыптасқан. Бұндай жалқаулықтың өзіндік себептері бар. Оны былай айтуымызға болады: «Программалар тез жазылады, бірақ олар баяу жұмыс істейді». Бұл келісім (ымыра) түрін ғажап деп атауға да болады. Өйткені адам сағатының құны компьютер сағатының құнынан әлдеқайда жоғары! Мысалы, Хоар алгоритміне құрылған программа Си тілінде Haskell тіліне қарағанда тез жұмыс істейді. Ол жұмыс атқару уақытына да, жадыны алу аумағына да қатысты. Бірақ Haskell тілінде жазылатын программа кодасы өте қарапайым. Екі программа да күрделілігі жағынан *O(nlogn)* деңгейінде. Яғни *n* ұзындықты тізімнің (массивтің) жұмыс істеу уақыты nlogn деңгейіне мөлшерлес түрде өседі [23].

Сонымен қатар функционалдық тілдерді трансляциялау алгоритмдері де тез дамуда, сондықтан олардың ассемблер кодалары императивті тілдер трансляторларына жетіп қалды деуге де болады.

Ал Haskell тілінің ең маңызды артықшылықтарына болашақта бұл тілдің трансляторларына қажетті алгоритмдерді қосу арқылы оның тиімділігін арттыруды жатқызуға болады. Мысалы, берілген функцияның анықтамасы бойынша өздері оларды есептейтін тиімді алгоритмдерді тауып, өздері оларды есептеп шығарады. Ондайда олар динамикалық программалау тәсілдері мен әдебиетте «сараң» деген атқа ие стратегияларды пайдалана алады. Қазіргі кезде алгоритмдер теориясының даму сатысы алгоритмдік есептер шаблонын тізбектейтін қатарларды бөліп қарап, функционалдық тілдер трансляторларын оларды берілетін функциялар анықтамасынан «танып» біліп, оларға тиімді есептеу алгоритмдерді қолдану дәрежесіне дейін үйретуге шамалары жетеді.

Әрине, кейбір жағдайларда Си тілін қолдану қажеттігі болады. Мысалы, жүйелік программалау, құрылғы драйверлерін жасау, сапалы графикасы бар, өте жоғары жылдамдықты қажет ететін компьютер ойындары сияқты қолданбаларға Си тілі қажет. Сонымен бірге көптеген компьютер қолданбаларына аса үлкен жылдамдықтың қажеті де шамалы болып келеді. Әдетте, оңтайландыруға қолданбадағы кейбір функциялар мен программаның кейбір бөліктері ғана жатуы мүмкін.

Функционалдық программмалау тілдерін көптеген күрделі жүйелерде қолдануға болады. Солардың кейбіреуін атап өтелік.

- *Software AG* – Германияның ең басты мықты программа құратын компанияларының бірі. Функционалдық программмалау тілінде *Natural Expert* деп аталатын сарапшы жүйесін құрған.

Көптеген пайдаланушылар осы жүйеге арналған өздерінің көптеген қолданбаларын аса ризашылықпен жазуда. Жүйе *IBM мейнфреймендерінде* жұмыс атқарады.

– *Ericsson* компаниясы Телефон станцияларын басқару жүйелерін құруға лайықты функционалдық *Erlang* тілін жасады.
– *MITRE* корпорациясының зерттеушілері Haskell тілін цифрлық сигналдарды өңдеуге пайдаланады.

Көптеген программалаушылар процедуралық тілдерде нысан бағытталған программалар құрып, ал нысан бағытталған тілдерде процедуралық стильдегі программалар жазады. Сол сияқты барлық программалау тілдерінде программалаудың функционалдық стилін қолдануға болады. Оның себебі – программалау тілдерін құрушылар өз туындыларына жан-жақты қасиеттер беруге тырысады. Бірақ есептердің барлық түрлерін шешуге жарайтын жалпыға бірдей тілді құру қиын. Бірақ осы мәселеде жеткен жетістіктер де бар. Соның бірі – *Python* тілі, ол көптеген программалау стильдерін қамтуға тырысқан және оның синтаксисі де қарапайым. Тілдің қарапайымдылығы сол тілде жазатын адам үшін қиындық келтіруі мүмкін. Мысалы, **Си** тілдерінде жазылған кейбір қарапайым программаларды компиляциядан кейін, онымен бір-екі сағат әуреге түскеннен соң ғана түсінуге болатын жағдайлар бар. Бұл программалаушы мамандар үшін, ал басқа адамдар үшін ол тіптен де түсініксіз болып қалады.

Сондықтан Haskell программалау тіліндегі шектеулерге құрметпен қарау қажет. Олар сізді көптеген қиындықтардан сақтап қала алады.

2.10. Лисп тіліндегі жаттығулар

Лисп тіліндегі жаттығуларды қарастырмас бұрын, Лисп тілінің функциялары мен предикаттарын қысқаша қайталап өтелік.

Лисп тіліндегі тізімдер рекурсивті құрылымда болады да, келесі түрде сипатталады:

```
(()(a b 1(c))class)
(45 89 (()()))
```

Бос тізім Лиспіде () және – ***nil*** деп белгіленеді. Кейбір S-өрнектерді есептеуге болады, бұндай өрнектер *формалар* деп аталады. Қарапайым формалар мысалы ретінде сандарды, ***T*** және ***nil*** атомдар-константаларды келтіруге болады. Қарапайым формаларға белгілі бір мәндер қабылдаған айнымалыларды да жатқызуға болады. Бұндай формалардың мәндері есебінде санның өзі немесе ***T*** және ***nil*** атомдарының мәндері немесе

айнымалының ағымдағы мәні жүре алады.

Форма ретінде функцияға қатынау түрін де қабылдауға болады, яғни келесі түрдегі тізім де жатады:

($f\ a_1\ a_2 \dots a_n$) ($n \geq 0$)

мұндағы f – функция аты, ал a_i – оның аргументтері, ол әдеттегі функциялар үшін формалар болуы да мүмкін. Сонымен бірге Лисп тілінде *арнайы функциялар* да бар. Олардағы аргументтер саны кез келген болады және олардың аргументтері не есептелмейді, егер есептелсе, оның есебі ерекше болады.

Лисп тіліндегі программа осындай формалар тізбегін құрайды және программаның орындалуы осы формаларды бірінен соң бірін есептейтін тізбектерден тұрады. Әдетте, программа басында жаңа функциялар анықталып, содан соң оларға қатынаулар басталады.

Лисп тілінде *кіріктірілген* немесе *ішкі* деп те атауымызға болатын функциялар көп. Осы функциялар негізінде пайдаланушы программасы құрастырылады. Төменде осындай *кіріктірілген* функциялардың кейбір түрлері келтірілген. Бұл функциялардың барлығы Лисп тілінің *Common Lisp* и *MuLisp* деп аталатын нұсқаларында бар [40].

Мынаны баса айта кетуіміз қажет: Лисп тілінде комментарий деп мына (;) таңбаның арасындағы мәтінде айтамыз.

Жаңа функцияларды анықтау.

Бұл мақсатты орындау үшін *defun* деп аталатын ішкі функция лайықты. Оған қатынау былайша орындалады:

(defun f (lambda($v_1\ v_2 \dots v_n$) e)) ($n \geq 0$)

(defun f ($v_1\ v_2 \dots v_n$) e)

Бұл функционалдық қалыптың мәні ретінде анықталатын функцияның атауы, яғни f жүреді. Бұл қалыпты есептеудің жанама нәтижесі ретінде v_i аргументтері (формальды параметрлерімен) және осы v_i параметріне тәуелді e қалыбындағы денесі бар f атаулы жаңа Лисп функциясын анықтауды келтіруге болады.

Жалпы жағдайда әдеттегі Лисп функциясын жаңадан анықтау жоғарыда аталған жолмен анықталады. Яғни тұрақты тіркелген аргументтер саны бар функция. Ол функция оған қатынағанда үнемі есептеулер жүргізеді.

Программада жаңадан анықталған функцияға қатынау былайша болады: *($f\ a_1\ a_2 \dots a_n$)* – мұнда ең алдымен a_i – функция аргументтері (нақты параметрлер) есептеледі, содан соң v_i жергілікті айнымалылар енгізіледі. Оларға әр айнымалыға сәйкес болатын a_i аргументтер мәні беріліп, одан ары қарай осы v_i айнымалы мәндеріне сәйкес e дене-

қалып есептеледі. Содан кейін бұл айнымалылар өшіріліп отырылады. Есептелген қалып мәні *f* функциясының мәні болады.

Тізімдерге қолданылатын операциялар.

(car l) – мұндағы *l* аргументінің мәні бос емес тізім болуы керек, ал функция мәні ретінде осы тізімнің жоғарғы деңгейіндегі бірінші элемент қабылданады.

(cdr l) – мұндағы *l* аргументінің мәні бос емес тізім болуы керек, ал функция мәні ретінде осы тізімнің «құйрығы» қабылданады, яғни бірінші элементі алынып тасталған тізім қабылданады.

Мысалы, егер *X* айнымалысының мәні болып *(p(q r))* тізімі табылса, онда *(car X)* қалыбының мәні р тең болады да, ал *(cdr X)* мәні *((q r))* тең болады.

Бұл екі функциядан басқа осы екеуінің қызметтерінің араласуынан туатын функциялар да бар. Олардың атаулары *a* әрпінен басталып, *r* әрпімен аяқталады. Олардың арасында төрт *a* және *d* әріптерінің комбинациясынан тұратын әрекеттер болады. Осы жағдайда суперпозициядағы *a* – әрпі *car* функциясының болуын білдірсе, ал *d* әрпі *cdr* функциясының болатынын білдіреді. Ал әріптер тізбегі суперәрекеттегі *car* және *cdr* функцияларының қалай алмасып келуін білдіреді. Мысалы: *(caddr l) ≡ (car(cdr(cdr l)))*.

Осы аталған суперпозициялық функциялардың *l* тізім-аргументтерінде өздеріне қажетті элементтер саны болады. Басқаша болған жағдайда программа үзіледі де, қате туралы хабар шығады. Тізім-аргументтерінде осындай қажетті элементтер саны болатын суперпозициялық функцияға *nth* функциясын да жатқызамыз.

(nth n l) – мұндағы *n* аргументінің мәні оң бүтін сан (оны *N деп белгілейміз*), болуы керек, ал *l* аргументінің мәні – тізім болуы қажет. Ал функция мәні ретінде осы тізімнің басындағы *N*-дік элемент болады. Және тізім элементтері тізім басынан 0-ден бастап нөмірленеді.

Мысалы, егер *X* айнымалысының мәні болып ***(p(q r)7)*** тізімі табылса, онда ***(nth 2 X)*** қалыбының мәні 7-ге тең болады да, ал ***(caddr X)*** мәні де 7-ге тең болады.

Жоғарыда қарастырылған барлық лисп-функциялары *селекторлар* деп аталады, өйткені олар тізімдегі белгілі бір элементтерді таңдап алады. Енді *конструкторлар* деп аталатын функцияларды қарастырайық. Олар өздерінің нәтижесі есебінде жаңа тізім құрады.

(cons e l) – бұл функция бірінші элементі ретінде *e* аргументінің мәні, ал тізім құйрығы ретінде *l* аргументінің мәні болатын тізімді құрайды.

(append l_1 l_2***)*** – бұл функция екі (жоғарғы деңгейдегі) тізім элементтерін қосылуын (конкатенациясын) іске асырып, одан бір нәтижелі тізімді алады.

(list e_1 e_2 ... e_n***)*** ***(***$n \geq 1$***)*** – бұл функциядағы аргументтер саны кез келген сан болады және аталған функция аргументтер мәндерінен тізім құрады. Соңғы нәтижелі тізімнің жоғарғы деңгейіндегі элементтер саны аргументтер санына тең.

(last l) – функция мәні болып бір элементтен тұратын тізім табылады. Ол *l* – аргументтер тізіміндегі жоғарғы деңгейдің соңғы элементі.

Мысалы, егер *X* айнымалысының мәні болып ***(p(q r))*** тізімі табылса және *Y* айнымалысы – тізім ***(t)*** болса, онда

(cons X Y) формасының мәні тең ***((p(q r))t)*** болады,

(list X Y) формасының мәні тең ***((p(q r))(t))*** болады,

(append X Y) формасының мәні тең ***(p(q r) t)*** болады,

(last X) формасының мәні тең ***((q r))*** болады.

Арифметикалық функциялар. Осындай функциялардың кейбіреуін қарастырайық. Мысалы:

(length l) – Бұл функцияның *l* аргументінің мәні тізім болуы керек. Функция осы тізімнің элементтер (жоғарғы деңгейдегі) санын есептейді. Мысалы, ***(length X)*** мәні **2**-ге тең, егер ***X*** айнымалысының мәні ***(p(q r))*** тізіміне тең болса.

Келесі функциялардың аргументтерінің мәндері сандар болады. Ол сандарға арифметикалық амалдарды қолдануға болады.

(add1 n) – Функция аргумент санына 1 санын қосып, оның нәтижесін өз мәні ретінде береді.

(sub1 n) – Бұл функция аргумент санынан 1 санын алып, оның нәтижесін өз мәні ретінде береді.

(+ n_1 n_2***)*** – Бұл функция мәні ретінде оның аргументтерінің мәндерінің қосындысы қабылданады.

(– n_1 n_2***)*** – Бұл функция мәні ретінде оның аргументтерінің мәндерінің айырмасы қабылданады.

(* n_1 n_2***)*** – Бұл функция мәні ретінде оның аргументтерінің мәндерінің көбейтіндісі қабылданады.

(/ n_1 n_2***)*** – Бұл функция мәні ретінде бір санның екіншісіне бөлгендегі бөліндісі қабылданады.

(rem n_1 n_2***)*** – Бұл функция мәні ретінде бір санның екіншісіне бөлгендегі қалдығы қабылданады.

Предикаттар. Предикаттар деп олардың логикалық мәндері «шын» немесе «өтірік» деген мәндерге ие болатын қалып (форма)

түрін айтамыз. Лисп тілінің ерекшелігіне «өтірік» мәніне ие есебінде бос тізім қабылданады да, ол мына ( ) таңбамен белгіленеді, ал «шын» деп осы ( ) және ***nil*** бөлек басқа кез келген өрнектер қабылданады. Мысалы «шын» рөлінде ***T*** атомы бола алады.

(null e) – бұл функция өз аргументі бос тізім бе, соны тексереді. Егер нәтиже оң болса, онда функция мәні ***T***-ге тең, ал керісінше болса ***nil***-ге тең.

(eq e_1 e_2***)*** – бұл функция өз аргументтер мәндерін салыстырады, олар атомдар-идентификаторлар болуы қажет. Егер олар тең (ұқсас) болса, онда функция мәні ***T***-ге тең, ал керісінше болса, ***nil***-ге тең.

(eql e_1 e_2***)*** – алдыңғы функциядан ерекшелігі – өз аргументтер мәндерін салыстырумен бірге, олар атомдар-идентификаторлар болуымен қатар атом-сандар болуын қадағалайды. Егер олар аргументтер тең болса, онда функция мәні ***T***-ге тең, ал керісінше болса, ***nil***-ге тең болады.

(equal e_1 e_2***)*** – бұл функция өз аргументтерінің мәнін кез келген S-өрнек мәнімен салыстырады. Егер олар аргументтер (яғни екеуі де бір S-өрнекті білдірсе) тең болса, онда функция мәні ***T***-ге тең, ал керісінше болса, ***nil***-ге тең болады.

(neq e_1 e_2***)*** – алдыңғы аталған функцияға ұқсас, бірақ аргументтер мәндері «тең емес» деген мағынамен салыстырылады.

(member a l) – бірінші аргумент мәні атом, ал екіншісінің мәні тізім болады. Бұл функция берілген атомды берілген тізімнің жоғарғы деңгейінде іздейді. Егер іздеу оң нәтиже берсе, онда функция мәні есебінде *l* тізімінің құйрығы беріледі, ал керісінше болғанда, ***nil*** тізімі беріледі.

Енді келесі берілген мына функцияларды қарастырайық.

$(= n_1\ n_2)$ $(/= n_1\ n_2)$

$(> n_1\ n_2)$ $(\geq n_1\ n_2)$

$(< n_1\ n_2)$ $(\leq n_1\ n_2)$

Бұл функциялардың аргументтері сандар болады. Бұл сандарға «теңдік», «теңсіздік», «көп», «көп не тең», «аз», «аз немесе тең» деген салыстырулар жүргізіледі, егер салыстырулар оң нәтиже берсе, онда ***T*** атомы қабылданып, керісінше жағдайда ***nil*** болады.

Логикалық функциялар. Негізгі логикалық операцияларды іске асыратын үш функцияны осы топқа жатқызады.

(not e) – Бұл функция логикалық терістеуді іске асырады. Оны ***null*** функциясының көшірмесі деуге де болады. Егер аргумент мәні ***nil*** («өтірік») тең болса, онда функция ***T*** («шын») деген нәтиже береді, ал аргументтің басқа мәндерінде нәтиже есебінде ***nil*** беріледі.

Келесі функциялар ерекше топқа жатады. Олар өз құрамында кез келген саны бар аргументтерді ұстайды және осы аргументтерінің мәндерін есептеу олар үшін міндетті емес.

(and e_1 e_2 ... e_k***)*** ***(k≥1)*** – Бұл функция конъюнкция операциясын іске асырады. Функция кезекпен өз аргументтерін есептейді. Егер олардың біреуінің мәні ***nil*** («өтірік») болса, онда функция басқа аргументтерін есептемей-ақ өз жұмысын ***nil*** нәтижесімен аяқтайды да, керісінше болған жағдайда келесі аргументінің мәнін есептеуге кіріседі. Егер функция осылайша соңғы аргументінің мәнін есептеуге жетсе, онда осы аргументтің мәнімен ол өз жұмысын аяқтайды.

(or e_1 e_2 ... e_k***)*** ***(k≥1)*** – Бұл функция дизъюнкция операциясын іске асырады. Функция өз аргументтерін кезекпен есептейді. Егер олардың біреуінің мәні ***nil*** («өтірік») болмаса, онда функция басқа аргументтерін есептемей-ақ өз жұмысын осы аргументтің мәнімен аяқтайды. Керісінше жағдайда келесі аргументінің мәнін есептеуге кіріседі. Егер функция осылайша соңғы аргументінің мәнін есептеуге жетсе, онда осы аргументтің мәнімен ол өз жұмысын аяқтайды.

Лисп функцияларының қатарына лисп *шартты өрнегі* деп аталатын өрнекті де жатқызамыз. Оның түрі мынандай болады:

$$(cond\ (p_1\ e_{1,1}\ e_{1,2} \dots e_{1,k1}) \dots (p_n\ e_{n,1}\ e_{n,2} \dots e_{n,kn})) \qquad (n \geq 1,\ k_i \geq 1)$$

Аталған функция кезекпен өз аргументтерінің бірінші элементтерін – p_i предикаттарын есептеп шығарады. Егер олардың барлығының мәндері ***nil*** («өтірік») болса, онда функция ***p*** жұмысын осы мәндермен аяқтайды. Егер ***nil*** мәнінен бөлек болатын p_i предикаты табылса, онда функция басқа предикаттарды қарастырмайды да, осыған сәйкес аргументтен (шартты өрнектің тармақтары) $e_{i,j}$ барлық қалыптарын бірінен соң бірін есептеп шығарады да, осы соңғы мәнмен өз жұмысын аяқтайды.

Тағы да мына жағдайды атап өтуіміз қажет: әр тармақтағы соңғы формалардан басқа қалыптар $e_{i,j}$ формасы есебінде пайдаланылады. Ал соңғы форма шартты өрнекті есептеу үшін қажет. Сонымен бірге қалыптарды пайдаланғанда, олардың жанама есептеулерді де орындауына көңіл аудару қажет. Мысалы, айнымалыға белгілі бір мән беру сияқты.

Басқа арнайы функциялар. Осындай функциялардың кейбіреуін қарастырайық. Мысалы:

(quote e) немесе ***'e*** – бұл функция есептеуді болдырмайды. Ол өз мәні есебінде есептелмеген аргументті береді. Мысалы, ***'(car (2))*** формасының мәні болып ***(car (2))*** өрнегінің өзі табылады.

(gensym) – бұл функция бірегей атомдарды-таңбаларды құрады. Оған әр кез қатынауда ол жаңа атом-идентификаторды беріп отырады. Бұл идентификатор арнайы префикс пен кезекті нөмірді (бүтін сан) бірбіріне жапсырудан пайда болады. Генерацияланатын атом нөмірлері басталатын префикс және бүтін санды алдын ала тағайындауға болады. Мысалы, MuLisp [40] тілінде:

(setq *gensym-prefix* 'S) (setq *gensym-count* 2)

Осыдан соң ***gensym*** функциясына әрдайым қатынағанда ол мына атомдарды ***S2, S3, S4*** және тағы басқа осы сияқты атомдарды беріп отырады.

Блоктық функциялар және олармен байланысты функциялар.

(prog $(v_1\ v_2 \ldots v_n)\ (e_1\ e_2 \ldots e_k)\ (n \geq 0,\ k \geq 1)$ – Бұл арнайы функцияны «блоктық» деп атайды, өйткені ондағы есептеулер басқа программалау тілдеріндегідей блоктардың орындалуын еске салады. Функция есептеулері былайша өтеді: жергілікті v_i айнымалылары енгізіледі, олар функцияның бірінші аргументінде анықталған және олардың барлығына бастапқы мән ретінде ***nil*** бос тізім мәні беріледі. Осыдан соң функция өзінің басқа аргументтері e_i формаларын есептейді. Осы қалыптардың ең соңғысын есептеген соң, ***prog*** функциясы осы форма мәнімен өз жұмысын аяқтайды. Оның алдына ол барлық өзінің жергілікті v_i айнымалыларын жояды.

Аталған ***prog*** функциясының мәні есебінде ең соңғы e_k формасының мәні пайдаланылатын болған соң, басқа e_i формалары үшін жанама әсері бар функцияларды пайдалану тиімді. Осындай функциялардың кейбіреуін келтірейік.

Аталған e_i қалыптарының бірі есебінде атом – идентификатор болады. Бұл жағдайда ол есептелмейді де, оны осы блок ішіндегі (***go*** функциясы) таңба есебінде қабылдайды.

(return e) – бұл функция блоктан кезектен тыс шығуды білдіреді. Ол блоктық ***prog*** функциясының ішінде қолданылады, өйткені жақын маңындағы блоктық функцияның есептеуін аяқтайды. Аталған блоктық функция мәні ***e*** аргументінің мәніне тең болады.

(go e) – бұл функция таңба (метка) бойынша өтуді іске асырады. Оның аргументі есептелмейді, оның аргументі есебінде идентификатор беріледі, ол – осы функцияға жақын орналасқан блоктык функцияның таңбасы бола алады. Осы ***go*** функциясы аталған блоктық функцияның

формасын есептеуді толығымен аяқтайды. Функция формаға (кез келген деңгейде) ене алады және таңбадан кейін көрсетілген форманы есептеуге кіріседі.

(setq v e) – бұл меншіктеу (присваивания) операторына ұқсас функция. Оның ***v*** (ол есептелмейді) аргументі есебінде осы сәтте бар болатын айнымалы атауы жүреді. Функция осы айнымалыға жаңа мән меншіктейді, ол – ***e*** қалпының есептелген мәні. Бұл мән ***setq*** функциясының да мәні болады, дегенмен, әдетте, ол көп пайдаланылмайды.

Келесі ерекше деп аталатын екі функция жиі пайдаланылатын мынандай ***(setq v(cdr v)) және (setq v(cons(e v))*** конструкцияларды жазуды оңайластыру үшін қажет.

(pop v) – бұл функцияның аргументі есебінде (ол есептелмейді) осы сәтте бар болатын және мәні бос емес тізім болатын айнымалы аты жүреді. Бұл бос емес тізімнің құйрығы айтылған айнымалының жаңа мәні болады және ол ***pop*** функциясының өз мәні ретінде де қабылданады.

(push e v) – бұл функцияның екінші аргументі есебінде (ол есептелмейді) айнымалы аты, ал бірінші аргумент есебінде кез келген форма қабылданады. Функция осы форманы есептеп шығарады және жаңа тізім құрады. Ол тізімнің бірінші элементі – есептелген мән де, ал құйрығы ***v*** айнымалысының мәнін қабылдайтын тізім болады. Алынған нәтижелік тізім ***v*** айнымалысының жаңа мәні және ***push*** функциясының мәні болады.

Мысалы, егер ***X*** айнымалысының мәні ***(d(e)g)*** тең болса, ал ***U*** айнымалысының мәні ***(1 2)*** тең болса, онда ***(pop X)*** қалыбының мәні ***((e)g)*** тең, ал ***(push U X)*** қалыбының мәні ***((1 2)d(e)g)*** тең болады.

Тізімдермен жұмыс істейтін кейбір мысалдарды қарастырайық.

1-мысал. Екі тізім берілген. Осы екі тізімді көбейту нәтижесін тізім түрінде алу қажет. Лисп тілінде программа кодасы мынандай болады:

```
(terpri)                ; Enter пернесіне ұқсас перне;
(terpri)
(terpri)
(terpri)
; proizvedenie функциясы анықталады;
(defun proizvedenie (spisok2 spisok1)
    (cond ((null spisok1) 0)                ; тармақталу;
    (T (prince (* (car spisok2) (car spisok1))) (prince " ")
```

```
(proizvedenie (cdr spisok2) (cdr spisok1)))))
(princ "Тізімді енгіз")
(terpri)
; a,b сандары енгізіледі;
(setq a (read))
(setq b (read))
(setq c () )                          ; c – бос тізім;
; Нәтиже экранға шығады;
(terpri)
(princ "Көбейту нәтижесіндегі тізім")
(terpri)
(proizvedenie a b)
(terpri)
(princ "Соңы !!!")
```

2-мысал. Екі сан берілген. Осы екі санның қосындысын табу қажет. Лисп тілінде программа кодасы мынандай болады:

```
; sum функциясы анықталады;
    (defun sum (N)
(cond ((=N 1) 1)              ; тармақталу;
    (T (+ N( sum (-N 1))))))
```

Одан ары қарай осы нәтижені экранға шығаруға болады. Ол үшін алдыңғы мысалдағыдай амалдарды кодаларға қосу қажет.

3-мысал. Тізім берілген. Осы тізім элементтерін қосудың нәтижесін табу қажет. Лисп тілінде программа кодасы мынандай болады:

```
; sl функциясы анықталады;
    (defun sl (L)
       (cond (( Null L) 0)         ; тармақталу;
          (T( + (car L))           ; тізім басы;
             (sl(cdr L))))))       ; тізім аяғы;
```

4-мысал. Сан берілген. Осы санның факториалын табу қажет. Лисп тілінде программа кодасы мынандай болады:

```
; fact функциясы анықталады;
   (defun fact (N)
     (cond ((= N 1) 1)             ; тармақталу;
        (T (* N(fact(-N 1))))))
```

5-мысал. Екі тізім берілген. Осы екі тізімнің бірінен соң бірінің алмасуын тізім түрінде беру қажет. Лисп тілінде программа кодасы мынандай болады:

```
; sher функциясы анықталады;
  (defun sher (x y)
   (cond ((Null x) y)                    ; тармақталу;
     (T (cons (car x)
        (sher y(cdr x))))))
```

6-мысал. Берілген N сандарының қосындысын табу қажет. Лисп тілінде программа кодасы мынандай болады:

```
; sum функциясы анықталады;
  (defun sum (N)
   (cond ((=N 1) 1)                      ; тармақталу;
     (T (+ N( sum (-N 1))))))
  (setq N (read))
 (terpri)
 (princ ' !!!')
```

7-мысал. 3 саны тізім түрінде берілген. Солардың ең үлкенін табу қажет. Лисп тілінде программа кодасы мынандай болады:

```
; sred функциясы анықталады;
(defun sred (x y z)
(cond ((> x y)                           ; тармақталу;
   (cond ((> y z) y)
        ((> x z) z)
        (t x)))
   (t(sred y x z ))))
```

8-мысал. Пілдер. Қойылымы: берілген 4x4 тақтасында 4 пілді олар бір-бірінің шабуылының астында болмайтындай етіп орналастыру қажет. Лисп тілінде программа кодасы мынандай болады:

```
; el_in функциясы анықталады;
(defun el_in (x y lst)
(cond
   ((NULL lst) 0)
   ((and (eql (car(car lst)) x) (eql(car(cdr(car lst))) y)) 1)
   (1 (el_in x y (cdr lst)) ) ))
```

```
; el_bit функциясы анықталады;
(defun el_bit (x y lst)
(cond
    ((NULL lst) 0)
    ((and (eql (abs (-(car(car lst)) (car(cdr(car lst))))) (abs(- x y)))) 1)
    (1 (el_bit x y (cdr lst)))))
; find_posl функциясы анықталады;
(defun find_posl (lst_el state)
(cond
  ((eql (length lst_el) 4) (print lst_el))
  (1
  (setq nx 0)
  (setq ny 0)
  (loop
  (setq nx (+ nx 1))
  (loop
  (setq ny (+ ny 1))
  (cond
  ((AND (eql (el_in nx ny lst_el) 0) (eql (el_bit nx ny lst_el) 0))
       (push nx state)
       (push ny state)
       (find_posl (cons(list nx ny) lst_el) state)
       (setq ny (pop state))
       (setq nx (pop state)) ) )
  ((> ny 3) (setq ny 0) NULL) )
  ((> nx 3) NULL) ) )))
; solve функциясы анықталады;
(defun solve()
(find_posl))
```

Лисп тіліндегі соқыр әдістердің жаттығулары. Соқыр әдістерді біз 3 бөлімде қарастырып өткенбіз. Осы топқа жататын бірнеше тәсіл жаттығуларының Пролог тіліндегі кодаларын да осы бөлімде келтіргенбіз. Енді осы тәсілдер алгоритмдерінің Лисп тіліндегі кодаларын қарастырайық. Бұл алгоритмдер [39] кеңінен келтірілген. Келтірілген алгоритмдер жайылып іздеу (BREADTH_FIRST_SEARCH) және шектелген тереңнен іздеу (LIMITED_DEPTH_SEARCH) түрін қамтиды. Екеуінде де бастапқы күй StartState деген атаумен белгіленеді. Екі тәсілде де Лисп функциясы өз мәні есебінде шешуші жолды (Лисп тізімі есебінде) табады немесе мынандай () бос тізімді анықтайды.

Енді бірінші тәсілдің Лисп тіліндегі кодаларын келтірейік. Келтірілген кодалар мәтінінде алгоритмге қатысты негізгі қадамдарға түсіндірмелер берілген. Төртінші және бесінші қадамдар программа кодасын қарапайым түрде карауға ыңғайластырылып біріктірілген. Программадағы келтірілген Open, Closed, Current идентификаторларының мәндері Лисп тіліне тән. Сонымен бірге қайталап пайда болатын күйлерді болдырмау үшін, программада RETAIN_NEW функциясы қарастырылған.

```
(defun BREADTH_FIRST_SEARCH (StartState)
  (prog (Open Closed Current
         Deslist                ; еншілес төбелер тізімі;
         Reflist                ; нұсқағыштар тізімі;
         Goal                   ; мақсатты төбе;
; 1 Қадам:; (setq Open (list (list 'S0 StartState)))
; 2 Қадам:; BS (cond ((null Open) (return ())))
; 3 Қадам:; (setq Current (car Open))
    (setq Open (cdr Open))
    (setq Closed (cons Current Closed))
; 4, 5 Қадамдар:; (setq Deslist (OPENING Current))
    (cond((setq Goal(CHECK_GOALS Deslist))
          (return (SOLUTION Goal Reflist)) ))
; Қайталанатын төбелер күйлерді болдырмау:;
    (setq Deslist (RETAIN_NEW Deslist))
    (cond ((null Deslist) (go BS)))
    (setq Open (append Open Deslist))
; Нұсқағыштарды құру және оларды жалпы тізімге енгізу:;
    (setq Reflist
          (append (CONNECT Deslist Current) Reflist))
    (go BS)))
```

Мысал кодаларында келтірілген қосымша функциялар да бар. Олар: OPENING, SOLUTION, IS_GOAL, CONNECT деп аталады және олардың мәні накты есептерге байланысты болатындықтан, олардың жалпы сипаттамасын бере кетейік.

Мұндағы OPENING функциясының негізгі қызметі өзінің бір аргументі үшін, ал оның аргументі есеп күйлерінің сипаты, – еншілес төбелер күйлерінің тізімін тудырады (ол тізім бос тізім болуы да мүмкін). Сипаттама әр күйді тізім арқылы береді, оның бірінші

элементі – күйдің бірегей идентификаторы, ол сан немесе атом болуы мүмкін. Ал екінші элемент есеп күйінің сипаты болып келеді. Бұл күйлер идентификаторлары CONNECT функциясының нұсқағыштарын құру үшін қажет.

Екі аргументі бар CONNECT функциясы өз құрамында пайда болған еншілес төбелер тізімі мен ағымдағы (жаңадан ашылған, аталық төбе) төбелер тізімін ұстайды. Бұл функцияның негізгі қызметі әр балалық төбеден оның аталық төбесіне бағытталған нұсқағыштар тізімін генерациялайды.

Екі аргументі бар SOLUTION функциясы өз құрамында табылған мақсатты күйлер тізімдері мен іздеу ағашында пайдаланылатын барлық нұсқағыштар тізімін ұстайды. Оның негізгі атқаратын қызметі есептің шешім тізімін (шешуші жолды) қалыптастырады.

Сонымен бірге кодаларда келтірілген IS_GOAL предикаты өзінің аргументінің мақсатты күйге жататынын тексереді. Егер тексеру оң нәтиже берсе, онда функция мәні есебінде тексерілген күйдің мәні қабылданады да, басқаша жағдайда бос мән () беріледі.

Тағы бір айта кететін жағдай жоғарыда аталған екі тәсілде де қолданылатын қосымша рекурсивті RETAIN_NEW функциясы жайында. Оның атқаратын негізгі қызметі – ол Dlist еншілес күйлер тізімінде тек бұрын пайда болмаған жаңа күйлерді ғана қалдырып, қалғанын өшіріп отырады. Сол арқылы кез келген өлшемді графтардағы циклге кетіп қалу қаупін азайтады. Енді осы функцияның кодаларын келтіре кетейік:

```
(defun RETAIN_NEW (Dlist)        ; қайту нүктелер тізімі;
(prog (D)
    (cond ((null Dlist) (return ()) )) ; тармақталу;
    (setq D (car Dlist))             ; тізім басы;
    (cond ((or (member D Open) (member D Closed)) ; тармақталу;
          (return (RETAIN_NEW (cdr Dlist))) ))     ; тізім құйрығы;
                  ; тізім басы мен құйрығынан тізім құру;
          (return (cons D (RETAIN-NEW (cdr Dlist))))))
```

Мысал кодаларында келтірілген тағы бір қосымша рекурсивтік функцияны түсіндіре кетейік. Программаларда келтірілген CHECK_GOALS функциясы балалық күйлер ішінде мақсатты күй бар-жоғын тексереді. Тексеру нәтижесі оң болса, бұл функцияның мәні

ретінде мақсатты күй қабылданады да, ал тексеру оң болмаса, онда бос тізім анықталады. Енді осы функцияның кодаларын келтіре кетейік:

```
(defun CHECK_GOALS (Dlist)
   (cond ((null Dlist) ())                    ; тармақталу;
         ((IS_GOAL (car Dlist)) (car Dlist))  ; тізім басы;
         (t (CHECK_GOALS (cdr Dlist))) ))     ; тізім құйрығы;
```

Енді екінші тәсілдің Лисп тіліндегі кодаларын келтірейік. Лисп тілінде анықталған LIMITED_DEPTH_SEARCH лисп-функциясы тереңнен іздеу деп аталады және оның екі аргументі бар: бастапқы күй – StartState және LimDepth деген алдын ала анықталған шектік тереңдік. Бұл аргументтер мынаны білдіреді: күйлер сипатын білдіретін әр тізімде күй идентификаторынан және есеп күйлері сипаттарынан басқа үшінші элемент болады, ол іздеу ағашындағы күйлер төбесінің тереңдігі. Программа кодасын қысқарту мақсатында алгоритм аяқталғанын (ол 6 қадамда орындалады) тексеру цикл ортасында іске асады. Тереңнен іздеу тәсілінің Лисп тіліндегі кодалары төмендегідей болады:

```
; Тереңнен іздеу функциясын анықтау;
(defun LIMITED_DEPTH_SEARCH (StartState LimDepth)
(prog (Open Closed Current
    Deslist        ; еншілес төбелер тізімі;
    Reflist        ; нұсқағыштар тізімі;
    Depth          ; ағымдағы төбенің тереңдігі; )
; 1 Қадам: бастапқы төбені енгізу (оның тереңдігі 0 тең):;
; ашылмаған төбелер тізіміне енгізу:;
        (setq Open (list (list 'S0 StartState 0)))
; 2 Қадам:;
LS (cond ((null Open) (return ())))
; 3 Қадам:; (setq Current (car Open))
        (setq Open (cdr Open))
        (setq Closed (cons Current Closed))
; ағымдағы төбенің тереңдігін анықтау:;
        (setq Depth (caddr Current))
        (cond ((IS_GOAL Current)        ;/ 6 Қадам /;
   (return (SOLUTION Current Reflist))))
; 4 Қадам:;       (cond ((eql Depth LimDepth) (go LS)))
```

```
; 5 Қадам: Төбені ашу, қайталанатын төбелерді алып тастау
және; нұсқағыш және ашылмаған төбелер тізімдерін жөндеу:;
            (setq Deslist (OPENING Current))
            (setq Deslist (RETAIN_NEW Deslist))
            (cond ((null Deslist) (go LS)))
(setq Open (append (ADD_DEPTH (add1 Depth) Deslist) Open))
(setq Reflist (append (CONNECT Deslist Current) Reflist))
            (go LS) ))
```

Тереңнен іздеу тәсілін Лисп тілінде бейнелейтін LIMITED_DEPTH_SEARCH функциясы жоғарыда келтірілген барлық қосымша функциялармен бірге тағы бір қосымша функцияны пайдаланады. Ол қосымша функция ADD_DEPTH деп аталады және оның атқаратын негізгі қызметі – ол Dn (тереңдік шамасы) санын Dlist тізіміндегі балалық күйлер сипаттарына қосады. Енді осы қосымша функцияның Лисп тіліндегі кодаларын келтірейік.

```
; Қосымша функцияны анықтау;
(defun ADD_DEPTH (Dn Dlist)
  (cond ((null Dlist) ())              ; тармақталу;
                ; тізім басы мен құйрығынан тізім құру;
(t (cons (list (caar Dlist) (cadar Dlist) Dn)
      (ADD_DEPTH Dn (cdr Dlist))))))        ; тізім құйрығына қосу;
```

Енді соқыр әдістердің тағы бір тобына жататын *эвристикалық іздеу алгоритмінің* мысалын қарастырайық. Біз 3-бөлімде осы топқа жататын алгоритм түрлерінің бірі «Тең бағалау әдісін» қарастырғанбыз. Тізім арқылы сипатталатын осы алгоритм жолдарының қысқаша сипаттамасын айта кетелік. Ол үшін алдымен осы алгоритмді іске асыратын Лисп тілінде HEURISTIC_SEARCH деген функциясы анықталады. Нақты іздеу есебіне тәуелді EST деп аталатын эвристикалық бағалау функциясын да анықтаймыз. Бұл эвристикалық баға әр күй үшін есептеліп, күйлерді сипаттау тізімінде төртінші элемент есебінде сақталады. Еске сала кетсек, жоғарыда келтірілген алгоритмдердегідей сияқты, тізімнің бірінші элементі – күйдің бірегей идентификаторы, екіншісі – күй сипатының өзі, ал үшінші элемент – іздеу ағашындағы осы күй төбесіне сәйкес келетін төбе терендігі. Тереңдік мәні эвристикалық бағаларды есептегенде жиі қолданылады, сондықтан EST функциясының аргументі ретінде екі элементтен

тұратын тізім жүреді. Яғни тізім ек элементтен тұрады: күй сипатының өзі және осыған сәйкес төбе тереңдігі. Мынаны атап өтуіміз керек: 3 қадамда, Open тізімінен бірінші элемент – төбені таңдап алғанда, ең аз (арзан) бағаны таңдаймыз, өйткені Open тізімі ондағы сақталған төбе күйлердің бағаларының азаюы бойынша тәртіптелген. Бұл тәртіпті Лисп тілінде қосымша анықталған MERGE функциясы қамтамасыз етеді. Енді сол тәсілдің Лисп тіліндегі кодаларын келтірейік.

```
; Эвристикалық іздеу функциясын анықтау;
(defun HEURISTIC_SEARCH(StartState)
   (prog (Open Closed Current
          Deslist        ; балалық төбелер тізімі;
          Reflist        ; нұсқағыштар тізімі;
          Depth          ; ағымдағы төбенің тереңдігі;)
; 1-қадам:;
   (setq Open (list(list 'S0 StartState 0
(EST (list StartState 0)) )))
; 2-қадам:;
   HS (cond ((null Open) (return())))
; 3-қадам:;
          (setq Current (car Open))
          (setq Open (cdr Open))
          (setq Closed (cons Current Closed))
          (setq Depth (caddr Current))
 ; 4-қадам:;
          (cond ((IS_GOAL Current)
          (return (SOLUTION Current Reflist))))
 ; 5-қадам:;
          (setq Deslist (OPENING Current))
 ; қайталанатын төбе-күйлерді алып тастау:;
          (setq Deslist (RETAIN_NEW Deslist))
          (cond ((null Deslist) (go HS)))
 ; 6-қадам:;
          (setq Open (MERGE (ADD_DEPTH_EST (add1 Depth)
          Deslist) Open))
 (setq Reflist (append (CONNECT Deslist Current) Reflist))
          (go HS) ))
```

Эвристикалық тәсіл бойынша іздеу алгоритмінде алдыңғы тәсілдердегідей (BREADTH_FIRST_SEARCH и LIMITED_DEPTH_SEARCH)

қосымша OPENING, SOLUTION, IS_GOAL, CONNECT, EST функцияларын пайдаланады. Сонымен бірге ол бұрын анықталған RETAIN_NEW функциясымен қатар тағы да екі қосымша – ADD_DEPTH_EST және MERGE функцияларын пайдаланады. Бірінші функция балалық төбелердің тереңдігін анықтаумен қатар, олардың эвристикалық бағасын да есептейді. Ал екінші функция екі тәртіптелген күй тізімдерін (эвристикалық бағалар мәндерінің төмендеуі бойынша) нәтижелі тәртіптелген тізімге келтіреді. Енді осы қосымша функциялардың Лисп тіліндегі кодаларын келтірейік:

```
; Қосымша ADD_DEPTH_EST функциясын анықтау;
(defun ADD_DEPTH_EST (Dn Slist)
    (cond ((null Slist) ())                          ; тармақталу;
; тізім басы мен құйрығынан тізім құру және сан қосу;
    (t (cons (list (caar Slist) (cadar Slist) Dn
          (EST (list (cadar Slist) Dn)) )
; Қосымша функцияның құйрығына қосу;
          (ADD_DEPTH_EST Dn (cdr Slist))))))

; Қосымша MERGE функциясын анықтау;
(defun MERGE (L1 L2)
    (cond ((null L1) L2)
    ((null L2) L1)
; тәртіптелуді қамтамасыз етеді;
    ((> (car (cdddar L1)) (car(cdddar L2)))
          (cons (car L2) (MERGE L1 (cdr L2))))
    (t (cons (car L1) (MERGE (cdr L1) L2))) ))
```

Лисп тілінде орындалатын жаттығуларға тапсырмалар

1. Мына қарапайым амалдарды Лисп интерпретаторы көмегімен орында:

а) 3.234*(45.6+2.43)
б) 55+21.3+1.54*2.5432-32
в) (34-21.5676-43)/(342+32*4.1)

2. мына өрнек мәндерін табу қажет:
а) '(+ 2 (* 3 5))
б) (+ 2 '(* 3 5))
в) (+ 2 (' * 3 5))

г) (+ 2 (* 3 '5))
д) (quote 'quote)
е) (quote 6)

3. Өз тобыңыздың студенттерінің тізімін құрыңыз (аты-жөні ... аты-жөні ... аты-жөні)

4. Әр студент үшін LIST функциясы көмегімен мынандай тізімдер құрыңыз: студент өзі үшін – (туған жылы), (мекен-жайы), (дәрістерден алған орташа баллы), (тәжірибе сабақтардан алған орташа баллы), (зертханалық жұмыстардан алған орташа баллы). Олардың ата-аналары үшін – (аты-жөні), (туған жылы), (мекен-жайы), (жұмыс орны).

5. Мына CONS және SETQ функциялары көмегімен жоғарыда аталған тізімдерді әр студентке қатысты таңбалар арқылы былайша жинақтаңдар: аты-жөні ст. – (((туған жылы ст.) (мекен-жайы ст.) ((дәрістерден алған орташа баллы) (тәжірибе сабақтардан алған орташа баллы) (зертханалық жұмыстардан алған орташа баллы))) (((әкесінің аты-жөні) (әкесінің туған жылы) (әкесінің мекен-жайы) (әкесінің жұмыс орны)) ((шешесінің аты-жөні) (шешесінің туған жылы), (шешесінің мекен-жайы), (шешесінің жұмыс орны)))).

6. Базалық функциялар көмегімен кез келген таңдап алынған студент үшін мына параметрлерді салыстырыңыз: а) туған жылы; б) оқу үлгерімі (орташа баллды сипаттайтын сан бүтін де, бөлшек те болуы мүмкін); в) туысқандық байланыс бар ма; г) ата-анасымен бірге тұра ма.

7. Әр студент үшін мынандай тізімдер құрыңыз а) дәрістерден алған бағасы), б) тәжірибе сабақтардан алған бағасы, в) зертханалық жұмыстардан алған бағасы.

8. Табиғи тілде берілген сөйлемді тізімге түрлендіретін функцияны жазыңыз.

9. Пайдаланушыдан топтағы студенттің аты-жөнін сұрау арқылы (топ тізімі бар деп есептеледі) ол туралы мынандай деректерді: туған жылы, орташа баллы, ата-анасы, оның басқа да қасиеттерін сипаттауды беретін функцияны жазыңыз.

10. Студенттер туралы тізім бар. Бір аргументі бар (студенттің аты-жөні), тізімнен орташа баллы бар тізімді шығаратын функцияны жазыңыз.

11. Студенттер жайында тізім бар. Бір аргументі бар (студенттің аты-жөні), тізімнен басқа тізімнен деректер алу арқылы орташа балды есептейтін тізімді шығаратын функцияны жазыңыз. Яғни екі тізімнен үшінші тізімді шығару.

Бақылау сұрақтары

1. Функционалдық программалау стилінің басқа стильдерден ерекшелейтін қасиеттерін атаңыз.
2. Функционалдық программалау тілдерінің өкілдерін келтіріңіз.
3. «Функцияларды байлау, біріктіру» тәсілін түсіндіріңіз.
4. «Программаларды байлау» тәсілі нені білдіреді?
5. «Жалқау есептеулер» деген не?
6. Haskell тілінің қасиеттерін атап өтіңіз.
7. Haskell тіліндегі айнымалылар мен функциялар айырмашылығы.
8. Haskell тіліндегі типтер мен жадыны басқару механизмі.
9. Haskell тіліндегі деректерді инкапсуляциялау ерекшелігі қандай?
10. Haskell тілінің қанша интерпретаторы бар ма?
11. Лисп тілінде дәрежелеуді көбейту мен бөлу амалы аркылы қалай анықтауға болады?
12. Лисп тілінде функция бір аргументте әртүрлі мән қабылдай ма?
13. Лисп тілінде қандай функция тізімнің соңғы элементін есептейді?
14. Лисп тілінде функционалмен берілген тізім элементерінен тізім құратын функцияны жазыңыз.
15. Лиспінің базалық функцияларын атап шығыңыз.
16. Базалық функциялардың аргумент типтері қандай?
17. Мына предикаттардың EQ, EQL, EQUAL және = негізгі айырмашылықтарын атаңыз.
18. Мына функциялардың: CONS и LIST негізгі айырмашылыктарын атаңыз.
19. Мына функциялардың: SET, SETQ, SETF негізгі айырмашылықтары қандай?
20. Лиспідегі таңбалар қасиеті қандай?

21. Лиспідегі тізімнің қай элементі басы, қайсысы құйрығы деп аталады?
22. Лиспідегі тізім элементтерін анықтайтын қандай функция?
23. Лиспідегі тізім қасиетін қайтаратын қандай функция?
24. Лиспідегі тізімдерге қатысты EQUAL предикатының орындалу нәтижесі қандай?
25. Лисп тізімінің физикалық құрылымын өзгертетін қандай функциялар?
26. Лисп тілінде қай функция тізімдегі атомдардың толық санын анықтайды?
27. Лисп тілінде лямбда-терм деген не?
28. Лисп тілінде қандай функция қарапайым таңдау алгоритмін пайдалана отырып тізімді іріктейді?
29. Лисп тіліндегі макростар қандай қызмет атқарады?
30. Лисп тіліндегі есептеу тармағын атқаратын функцияны атаңыз.
31. Лисп тілінде қандай функция тізім элементтерінің циклдік алмастыруын орындайды?

3-бөлім. ЖАСАНДЫ ЗЕРДЕ ЖҮЙЕЛЕРІ

Жасанды зерде мен ондағы проблемалар. Жасанды зерде (ЖЗ) – деп саналы тәртіпті басқаруды автоматтандыратын компьютерлік ғылымның аумағын атаймыз. Жалпы, «Зерде-Интеллект» деген не? Яғни оны қанша мөлшерде құруға болады және ол қандай мөлшерде алдын ала табиғатта бар? «Шығармашылық» деген не, түйсіну деген не, интеллект бар екенін байқап қадағалап көреміз бе, әлде оны басқаратын жасырын күш бар ма? Тірі заттардың нерв клеткаларындағы білімдер қалай құрылған және зерделік құрылғыларды жобалағанда, оны қалай қолдануға болады? Өзіндік талдау деген не және ол саналықпен қалай байланысты болады? Зерделік компьютерлік программаларды адам санасына ұқсас құру керек пе, әлде қатаң түрде «инженерлік» жол жеткілікті ме? Саналылықты компьютер техникасы көмегімен құруға бола ма? Әлде зерде мағынасы тек биологиялық жәндіктерге тән сезім мен тәжірибе арқылы ғана анықтала ма? Осындай көптеген сұрақтар төңірегіндегі зерттеулер адам баласын көптен бері толғандыруда [24].

Көне Грекиядағы адамдар үшін Олимп құдайлары отты ғана ұрламай, олардың ақыл көздерін ашты, яғни білім бұлағына жақындатты. Бұрын тек Олимп құдайларына ғана белгілі білімдерге адамдарды жақындатқандығы үшін, Зевс құдай Прометейді тас жарқабаққа мәңгілікке байлап қойды. Әлі күнге дейін Батыс философиясында адам баласының білімге деген құштарлығы мен оған тереңдеуі құдайлар алдындағы күнәлі әрекет деген ойлар белең алады. Осындай сенім ойлары Қайта өрлеу дәуірінде, ғылыми, философиялық ашылымдар жасаған 19 және 20 ғасырларда да болды. Сондықтан қазіргі кездегі ғылыми және қоғамдық көзқарастардағы ЖЗ жайындағы қызу таластар тоқтамағаны да – заңды нәрсе.

Иә, сонымен, ЖЗ саласының философиялық түбірінің бірнеше мың ғасырлық тарихы бар екен. Бұл тарихтың бастапқы қойнауының бірінде Аристотель тұр. Өзінің «Физика» атты еңбегінде ол: «Табиғат философиясы – өзгеріп тұратын заттарды зерттеу», – деп атады. Ол зат және форма арасындағы айырмашылықты зерттеді.

Мысалы, мүсін өнерін алсақ, заттар қолдан жасалған, ал формасы адам қалпында. Өзгеру қолаға басқа форма бергенде болады. Материя мен форманы осылайша бөлу танбалық есептеу немесе деректер абстракциясы сияқты ғылыми тұжырымдама негізін салады десе де болады. Кез келген есептеулерде біз электромагниттік материяның формасы болатын бейнелермен жұмыс істейміз, ал осы форма өзгеруі шешу

үдерісінің бір қырын береді. Форманы оны бейнелетін құралдардан бөліп қарау қазіргі заманға компьютер ғылымының ядросы болып табылатын зеректік құрылым теориясының негізін құрайды.

Өзінің «Метафизика» атты еңбегінде Аристотель өзгермейтін заттар теориясын да жасаған. Бұл заттарға ол космология және теологияны жатқызады. «Логика» атты еңбегінде ғалым ой пайымдауларының басқа да ой тұжырымдарымен байланысын зерттей отырып, осы пайымдаудың дұрыс екенін дәлелдейтін мәселелерді қарастырған. Мысалы, егер екі түрлі тұжырым бар болса: «барлық адамдарға ажал бар», «Сократ-адам» және олар дұрыс болса, онда «Сократқа ажал бар» деген тұжырым жасауға болады. Бұл силлогизм мысалында modus ponens - деп аталатын дедуктивтік ереже пайдаланылған. Ойлау мен сананың қазіргі тұжырымдамасындағы Рене Декарттың жасаған тұжырымдары орталық орын алуда. Ол өзінің «Размышления» атты еңбегінде нақты нәрсе негізін «Когнитивтік интроспекция» әдісі деп аталатын жолмен бағалауды ұсынды. Сезім органдарымен түсетін ақпаратты сенімді деп тапқан ол тек ойлар ғана нақты дүниені білдіреді деп есептейді [9]. Бұл философиялық ойлардың ЖЗ үшін маңыздылығы мынада:

1) Декарт және оны жақтаушылар дүниені сана және физикалық дүние деп екіге бөліп қарағанда, мынандай қорытындыға келді – дүниедегі құрылым жайындағы идея оқылатын пәнге үнемі сәйкес келе бермейді. Осы пікірде ЖЗ әдіснамасы да құрылған.
2) Дене мен Сана бөлек деп саналғандықтан, философтар оларды біріктіріп, бір-біріне әсері болады деген тәсіл ойлап тапты. ЖЗ ғылымының бағыты да осындай тәсіл түрлерін оқып зерттейді.

3.1. Жасанды зердеің негізгі ұғымдары

Логиканың дамуы. Орта ғасырдағы ғалымдар ойлаудың өзін есептеулер қалыбы деген пікірде болды. Сондықтан бұл үдерісті зерттеудің нәтижесінде ойлаудың өзін алдымен қалыпқа келтіру (формализация), сосын механикаландыру (механизация) қадамдарын істеу арқылы басқаруға бола ма деген пікірлер туа бастады. 18 ғасырда *Готфрид Вильгельм фон Лейбниц* өзінің «Calculus Philosophicus» деген еңбегінде ең алғашқы формальды логиканың жүйесін жасап, оның есептеулерін автоматтандыратын машина ойлап тапты. Нақты дүниедегі шешілетін мәселелерді бейнелеуге бағытталған құрал есебінде швейцар математигі *Леонард Эйлер* бейнелеу ілімін құрды (учение о представлениях). Ол – қазір классикаға айналған графтар теориясы. Бұл

теория есеп құрылымы мен күрделілігін талдаумен бірге оны шешу амалдарын да береді. Эйлер бұл теорияны «Кейнсберг көпірлерінің есебі» деп аталатын есепті шешу үшін ойлап тапты. 19 ғасырда да жасанды зерде бағытындағы зерттеулер толастаған жоқ. Математик *Чарльз Бэббидж* «Разностная машина» деп аталатын машина құрды. Математик өз өмірінде жобалап құрған бұл сараптамалық машина сипатын *Ада Лавлейс* былайша жазды: «Сараптамалық машина, Жаккард станогі гүлдер мен жапырақтардан қандай кестелер тоқыса, бұл машина алгебралық кестелер тоқиды». *Бэббидж*дің сараптамалық машинасы жады мен процессорды бөліп қарау идеясына негізделді. *Бэббидж* терминологиясында олар «қойма» және «диірмен» деп аталды. Онда программалау принципінде цифрлық есептеулерге, яғни операцияларға көңіл бөлінеді. Бұл цифрлар мен операциялар кодалары картон перфокарталарында сақталады. *Бэббидж* жұмысында ең алдымен сонау Аристотель сипаттап кеткен «Абстракция және қалыпты өзгерту» принципін іске асыру болған. Яғни алгебралық кестелер байланыстарын «мән» деп қарау. Оларды зерттеп, сипаттап, іске асырып, ең соңында механикалық іс-шаралардан өткізу. Осы іс-әрекеттердің барлығы бұл «мәндердің» қандай мағына алуына байланысты емес, ол есептеу үдерісінде есептеу машинасы «диірменінен» өтуге тиісті болды [10].

19 ғасыр математигі *Джордж Буль* да, ойлау үдерісін сипаттайтын формальды тіл құруға әрекет жасады. Ол логика заңдарының математикалық қалыбын жасады. Бұл қалып қазір де компьютерлік ғылымдардың нағыз жүрегі десек те болады. Д. Буль «Исследование законов мышления, на которых основывается математическая теория логики и вероятности» деген еңбегінде өз мақсатын былай жазады: «Тұжырым жасау сияқты сана операциясының іргелі заңдарын зерттеу. Оған есептеудің таңбалық тілінде өрнектер беру және осының негізінде логика ғылымын орнатып, осы логика әдістерін үйрету. Түптің түбінде осы тұжырымдарға әртүрлі шындық элементтерінің көмегімен адам миының ойлау табиғаты мен сақталу құрылысы жайлы мүмкін болатын ықтималды ойлардың қорытындысын жасау». Буль еңбегінің ұлылығы оның ойлап тапқан жүйесінің қарапайымдылығында. Оның логикалық есептеулерінің негізі үш логикалық операциядан тұрады: «және-И-(* не ^)», «немесе-ИЛИ-(+ не v)», «емес-не-(-|)». Бульдың операциялары екі сандық мәнмен жұмыс істейді: 1 және 0. Бульдың жүйесі екілік арифметика негізін қалады. Буль жасаған жүйе бүкіл логиканың

қалыбын жасау жұмыстарына классикалық негіз болды десек, артық емес.

Арифметика негіздерін сипаттайтын өзгешеліктердің анық және дәл тілін *Готлоб Фреге* құрды. Ол өзінің «Арифметика негіздері» атты еңбегінде Аристотельдің «Логика» еңбегінде қаралып өткен көптеген сұрақтарды сипаттайтын тіл құрды деуге болады. *Фреге* тілі қазір «І-дәрежелі претикаттарды есептеу» деген атпен белгілі. *Фреге* тілі математикалық тұжырымдар элементтерін құратын теоремалар жазуға және олардың шындық мәндерін жазуға арналған таптырмас құрал болды. Өз құрамында предикаттар таңбалары, функциялар және кванталған айнымалылар теориясын ұстайтын осы предикаттарды есептейтін формальды жүйе математика және оның философиялық негіздерін сипаттауға арналған тіл бола алады деген болжам да болды. Дегенмен бұл құрал ЖЗ есептерін бейнелеу теориясын құруда зор рөл атқарды. Математикалық тұжырымдары таза формальды тұрғыда қарастыру *Рассель* мен *Уайтхед* еңбектерінде зерттелді. Бұның ЖЗ-нің іргелі принципі үшін маңызы зор болды. Бұл ғалымдар математика ғылымын таза формальды жүйе деп есептеді. Яғни аксиомалар мен теоремалар – таңбалар жиынтығы есебінде қарастырылады. Ал дәлелдеу осы жолдарға қатаң түрде белгілі бір ережелерді қолдану нәтижесінде іске асады. Осындай логикалық синтаксиспен шығарудың формальды ережелері теореманы автоматты түрде дәлелдеу жүйелерінің негізін құрайды. ЖЗ қалыптасуында тағы бір маңызды рөл атқарған математиктің бірі – *Альфред Тарский*. Ол сілтемелер теориясын құрды. Ол теория бойынша *Фреге* немесе *Рассель Уайтхед* құрған ППФ (Правильно построенная формула) – ДҚФ (Дұрыс құрылған формула) (well formed formulae) негізінен дүниедегі бар нысандарға сілтеме жасайды. Олар «Семантическая концепция», «Истинности и основание», «Семантики» деп аталатын еңбектерінде сілтемелер теориясын сипаттайды.

Цифрлық компьютерлердегі машинаға арналған ең алғашқы зерттеулерді британ математигі *Алан Тьюринг* жүргізді. Ол 1950 жылы «Вычислительная машина и интеллект» деген еңбегін Mind журналында жариялады. *Тьюринг:* «Машинаны ойлауға үйретуге бола ма?» – деген сұрақты зерттеді. Ол бұл сұрақтың өзінде белгісіздік бар деп есептеді. Яғни ойлау деген не? Машина деген не? Сұрақтардың өзіне жауап іздеу керек. Сондықтан ол интеллект деген ұғымды эмпирикалық тест тұрғысынан түсіндіруді жөн деп санады. Осылайша қазіргі заманда да өз маңызын жоймаған Тьюринг тестісі деп аталатын тест жасалды. Енді осы тестіге қатысты мәселелерді шолып өтелік [9].

Тьюринг тестісі. Бұл тест ақылды машина қабілеті мен адам қабілетін салыстыруды өткізеді. Тестіні «Еліктеу ойыны» деп атаған *Тьюринг* машина мен адамды (оны тергеуші деп атаған) әртүрлі бөлмелерге орналастырған. Машинаны имитатор (еліктеуші, ұқсатушы) деп атаған.

Тергеуші имитатормен тек терминал арқылы байланысады, ол оны көрмейді, естімейді. Тергеуші компьютерді адамнан тек оның берген жауабы арқылы ғана ажырата алады. Егер тергеуші компьютерді адамнан ажырата алмаса, онда Тьюринг пайымдауы бойынша машинаның санасы бар деп есептеуге болады.

Бұл тестің мынандай маңызды қасиеттері бар:

1) Интеллект – зерде ұғымы туралы нақты түсінік береді, яғни ол дегеніміз – саналы объектің белгілі бір сұрақтар жиынтығына беретін әсері, яғни реакциясы осылайша болатын «Интеллект» ұғымы анықталады.
2) Түсініксіздік тудыратын сұрақтарды болдырмауға әрекет жасайды. Ондайларға компьютер ойлау үшін өзінің ішкі әрекеттерін пайдалана ма, әлде машина өзінің әрекеттерін саналы түрде жасай ма деген сұрақтар жатады.

Бұл сұрақтардың барлығына толық жауап әлі жоқ, дегенмен осы сұрақтар төңірегінде өрбіген мәселелер мен оның әдістемелік жолдарын зерттеу қазіргі заманғы жасанды зерде ғылымына қатысты саланың негізін құрды. Жасанды зерде осы уақытта дейінгі адам баласы жасаған технологиялар мен өркениеттер негізі болады.

Зерделік жүйелердегі табиғи тілді түсіну жолдары мәселе шешілетін аумақтағы білімдерге тікелей байланысты болады. Тілді түсіну деген ондағы сөздер мен сөйлемдерді айту ғана емес. Ол үшін аталған тілде сөйлеп тұрған адам сала мәселесінің түсінігін де сол тілде жеткізе білуі керек. Табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдердің анықталған пайдалану құрылымдарын қажет етеді. Осындай құрылым түрлерін жасағанда, білімдердің өзгешілігін білдіретін қасиеттерімен бірге, адамдар арасында болатын күрделі қарым-қатынастар түрлерін, бір сөздің бірнеше мағынада айтылуы, үйрену әдістері, әртүрлі көзқарастар сияқты мәселелердің барлығын есепке алу қажет.

Табиғи тілді түсіну мәселесін шешу үшін көптеген сұрақтарға жауап табу керек. Біріншіден, адам білімінің үлкен ауқымды көлемі қажет. Табиғи тілді түсінуді іске асыратын компьютерлік жүйеде күрделі де нақты дүниедегі шым-шытырық байланыс түрлері сипатталуы қажет.

3.2. Есептерді шешудің түрлі әдістері

Бұл параграфта Есептерді шешудің эвристикалық әдістерінің әдістемелік негіздері, Кеңістіктегі күйі және редукция әдістерімен Есептерді шешу жолдары қарастырылады [27].

Шешімді іздеу тәсілдері мен іс-әрекеттерін талдау. Білімді пайдалану үлгісін таңдап алып, сол бойынша білімдер құрылымын құрған соң, осы саладағы мәселе бойынша есептердің шешімін табу қажеттігі туады. Ол үшін іс-әрекеттің белгілі бір жоспары қажет. Мұндай жоспар әдеттегі математикалық белгілеуде алгоритм деп аталған. Ал зерделік жүйелер деп аталатын программаларда алдын ала дайындалып қойылған іс-әрекет жоспары болмайды. Ол жоспар программа іске қосылған соң жағдайға сәйкес құрылуға тиіс. Қазіргі кезде ондай жоспар құру үшін қолданылатын әдістер екі топқа бөлінеді: кеңістіктегі күйі бойынша жоспарлау, кеңістіктегі есептер жағдайы бойынша жоспарлау. Осы әдістерге шолу жасап өтелік.

Кеңістіктегі күйі бойынша жоспарлау. Бұл жағдайда кеңістікте белгілі түрде анықталған жай-күй берілді деп есептейміз. Бұл жай сыртқы дүниемен зерделік жүйені белгілі бір шамалармен сипаттайды. Берілген ахуалдар сыртқы дүниенің жалпыланған күйін білдіреді де, ал зерделік жүйедегі программалардың іс-әрекеті осы берілген жайдың өзгеруіне әкеліп соғады. Сыртқы дүниенің жалпыланған жағдайлары ішінен оның ең алғашқы күйі деп аталатын (ол әдетте біреу), ең соңғы күйі деп аталатын жағдайларды бөліп қарауға болады. Бұндайда іс-әрекетті жоспарлау мақсаты – алғашқы күйден нысанаға дейінгі жеткізетін жолды табу. Есептерді кеңістіктегі күйі бойынша жоспарлағанда, оның мынандай жағдайлары бейнеленуі қажет: жай-күйі, операторлар жиыны, сол операторлардың ахуалға ететін әсері. *Соқыр әдіс.* Бұл әдістің екі түрі болады: тереңнен іздеу, жайылып іздеу. Тереңнен іздеуде әрбір күй аяғына дейін тексеріледі де, басқа күйлердің әсері есепке алынбайды. Тереңнен іздеу жолын өте биік болып келетін ағаштар үшін қолдану тиімсіз. Өйткені керек төбенің қасынан ұзап кетеді де, басқалардың бәрінен өту үшін көп уақыт кетеді. Жайылып іздеуде барлық күйлер ең төменгі жағында орналасқан болса, бұл әдіс жолын қолдану тағы да тиімсіз болады. Бұл әдістің екі жолы да көп уақытты керек ететіндіктен, бағытталған әдістерді қолдану қажеттігі туады. *Бағытталған әдістер.* Мұндай әдістер қатарына шекара және тармақ әдісі, Мурдың ең қысқа жол алгоритмі, Доран мен Мич алгоритмі, Харт, Нильсон, Рафел алгоритмдерін жатқызамыз. Шека-

ра және тармақ әдісінде шешу барысында табылған, әлі аяқталмаған жолдың ең қысқасы таңдалып алынады. Осы жол бір қадамға ұзартылады, жаңадан пайда болған аяқталмаған жолдар (олардың саны графтағы осы төбеге қатысты тармақтар санымен бірдей), қалған жолдармен қосылып олардың ішінен ең қысқасы таңдап алынады да, ол бір қадамға ұзартылады. Бұл үдеріс бірінші нысаналы төбеге жеткенге дейін қайталанып, осы шешім еске сақталады. Ары қарай қаралмаған тармақтағы жолдар да осы әдіспен тексеріледі. Ең аяғында аяқталмаған жолдар бітеді немесе олардың арасында алдыңғыға қарағанда нысанаға жетудегі ең қысқасы қалады. Бұл шешім жолы болып табылады. Мурдың ең қысқа жол алгоритмінде ең алғашқы төбе болып есептелетін X_0 төбесіне 0-мәні беріледі. Кез келген X_i төбесінен таралған еншілес төбелер жиынын $Г(x_i)$ деп белгілейік. Сонда бұл жиыннан алғашқы кездескен төбелердің бәрі сызылып, қалғандары белгіленеді. Және де ол X_i белгісінен бір санға артық белгіленеді. Осы белгіленген төбелер ішіндегі белгі саны ең азы таңдап алынып, осы төбеден ары қарайғы оның еншілес төбелері ізделіп, сосын олар да белгіленеді. *Кеңістіктегі есептер жағдайы бойынша жоспарлау.* Бұл жоспарлау берілген есепті ең қарапайым түрге келтіргенге дейін бөлшектеп, бөлектеп қарауға болады деген ойға негізделген. Осы бөлшектелген әрбір есептің шешімдер жиыны берілген есептің шешімі болып табылады. *Кілт операторы әдісі.* Кез келген *А, В* есебі берілген деп есептейік. Бұл есепке f операторы қолданылған. Осы f операторын қолдану нәтижесінде f(С) күйін алдық делік. Сонда ЖӘНЕ төбесі *А, В* үш балалық төбеге таралады: *А, С*, *С, f(С)*, *f(С), В*. Соның ішінде ортаңғысы қарапайым есеп болып табылады. Қалған *А, С*, *f(С), В* есептеріне тағы кілт операторын қолдану арқылы оларды да қарапайым есептерге бөлшектеуге мүмкіндік туады. Сонымен, ең аяғында *А,В* есебін бірнеше қарапайым есептерге бөлшектеуге болады. Мұнда бір қиындық келтіретін жағдай – қажет болатын кілт операторын табу мүмкіндігі. Жалпы жағдайда А мен В арасындағы айырмашылық есептеліп, осы айырмашылықты жоюға мүмкіндік беретін операторлар ізделеді.

Жалпылама шығару әдісі. Бұл әдіс іздеу принциптерінің екі түрін біріктіреді: мақсат арқылы құралды талдау, есептің рекурсивтік шешімін табу. Бұл әдіс іздеудің әрбір қадамында қатаң тәртіппен үш түрлі есепті шешеді: А нысанын В нысанына өзгерту, А мен В арасындағы айырмашылықты білдіретін D-ның мөлшерін азайту, f операторын А-ға қолдану. Бірінші есептің шешімі – D-ны табу, екінші

есептікі – f – операторын табу, үшіншінікі – С күйін қолдану шартын табу. Егер С мен А бірдей болса, онда f операторы қолданылады, өзгеше болса, С келесі нысана есебінде алынады да, есеп ары қарай қайталанады, яғни «А-ны С-ға өзгерту» мақсаты орындала бастайды. Жалпы жағдайда бұл әдіс *А, В* есебін *А, С* есебіне және *С, В* есептеріне бөлшектеу бойынша өтеді.

Білімдерге дедуктивтік қорытынды шығару. Білімді пайдалану мен өңдеуде дедуктивтік үлгіні қолдану үшін шешілуге тиісті мәселе формальдық жүйенің пікірі есебінде жазылуға тиіс. Яғни мақсат та осы жүйе тілінде пікір есебінде жазылып, оның дұрыс-бұрыстығын аксиомалар және шығару ережелері көмегімен дәлелдеу қажет. Формальдық жүйе есебінде бірінші дәрежедегі предикаттар есептеуі тілін алуға болады. Бұл формальды түрдегі қабылдап алынған ережелер заңы бойынша, ең соңғы пікір – теоремаға «шын» деген мағына дұрыс болған жағдайда ғана қабылданады. Егер кез келген жағдайда F_1, F_2, ..., F_n формулаларының конъюнкциясы, яғни $F_1\$ F_2\$, ..., F_n\$$ «шын» мағынада болса, онда В формуласы F1, F2, ..., Fn формуларының логикалық сандары болып табылады. Мұндай теореманың дәлелдеуі деп мынандай сұраққа жауап іздеуді айтамыз: В-формуласы F_1, F_2, ..., F_n жиынның логикалық салдары болып табыла ма? Немесе мынандай формуланы дәлелдеу қажет: $F_1\$ F_2\$, ..., F_n\$ \rightarrow B$. Іс жүзінде соңғы формуламен жұмыс істеу ыңғайлы. Онда да ол формуланың жалғандығын дәлелдеу қажеттігі туады. Мысал үшін білімдер семантикалық тор үлгісінде берілгенде қолдануға болады.

Жалпы жағдайда фактілер мен жалпылама зандылықтардан (аксиомалар, шарттар) тұратын семантикалық тор – бір саладағы мәселенің хабарлар үлгісі болып табылады. Тордың төбесінде нысандар немесе ұғымдар, ал оларды қосып тұрған сызықтар – доғалар, олардың арасындағы байланыстарды білдіреді. Әрбір дизъюнктар жиынында, ал жалпы дизъюнктар түрі $B_1\$ B_2\$ ... B_m\$ \rightarrow A_1 V A_2 V ... A_n$ немесе $B_1\$ B_2\$ ... B_n\$, A_1 V A_2 V ... A_n$ болып келеді. Мұндағы B_i – дизъюнкт шарты, A_j – соңы. Шарты да, соңы бар дизъюнктер жалпы зандылықты білдіретін аксиомаларды көрсетумен қатар, бұл білімнің интенсионалдық көрінісін де бере алады. Шартсыз дизъюнктер фактілерді көрсетеді, яғни ол – білімнің экстенсионалдық көрінісі. Соңы жоқ дизъюнктер - ол фактілерді жоққа шығару немесе дәлелдеуге тиісті мақсат-нысаналар. Дедуктивтік қорытындымен шығарылатын есеп былайша қойылады: семантикалық тор түріндегі логикалық лингвистикалық үлгі берілсе, онда осы торға сәйкес келетін дизъюнк-тер жиыны да беріледі.

Қиысқан жол әдісі. Бұл әдіс семантикалық торларда көп қолданылатын әдістер қатарына жатады. Бұл әдіс семантикалық торлар төбелері белгілі бір кілттер арқылы берілсе және олар арасында анық байқалатын байланыс түрі болса, онда осы кілттерді бейнелейтін пікірдің дұрыстығын дәлелдеуге болады деген ойға негізделген. Ол үшін тордың екі төбесінен (мысалы, A, B) барлық жаққа қарай іздеу жүргізіледі. Ол іздеу A төбесінен шыққан жолмен (1a) B төбесінен шыққан жол (1b) қиылысқанша жүріп отырады. Әдетте, 1a+1b қосындысы ең аз болатын төбе таңдалып алынады.

Бір-біріне жапсырылған жол әдісі. Бұл әдісте семантикалық тордың кейбір бөліктері алынып, бір-бірімен салыстырылады. Яғни салыстыру нәтижесі оң болса, семантикалық тордағы іздеу сонша бөлікке оңайланады.

Әдейіленген қорытынды жасау әдісі. Кейбір жүйелерде әдейіленген ережелер қолданылады. Бұл әдіс – ұғымдар арасындағы байланыстар қасиетіне негізделген. Мысалы, егер R байланысының транзитивтік қасиеті болса, aRb фактісінің шындығын дәлелдеу үшін, a мен b арасындағы R байланысы бар жолды табу қажет. Егер мұндай жол табылса, aRb фактісінің шын екені дәлелденеді. Жалпы жағдайда R байланысын басқа байланыстармен қарым-қатынас түрінде қарайды: $R = R_1\ R_2\ ...\ R_n$, және бұл қарым-қатынас түрлері іздеу жолы ретінде қаралады.

Боялған семантикалық тордағы дедукция алгоритмі. Мысалға боялған семантикалық тор үлгісін қарайық. Енді осындай торға дедукция алгоритмін қолдануды қарастырамыз. Бұл алгоритмде: P деген предикат төбесі үшін екі түрлі литера таңдап алынады (доғалар әртүрлі түспен боялған). Осы таңдап алынған литера үшін «бос тор» шығару жолы қарастырылады. Егер төбеге бір түспен боялған доғалар кірсе, олар іздеу барысынан шығып қалады. Бұл дедукциялық алгоритмнің ерекшелігі – семантикалық тор түрі өзгермейді.

Жою операторы мен тізбелерді бөлуді қолданатын дедукция алгоритмі. Бұл алгоритмді қолданғанда, семантикалық тордың түрі өзгеріске ұшырайды. Ол үшін төбелерді жою операторы мен оларды бөлу операторы қолданылады. *Жою операторын қолдану.* Ең алдымен мультидоғаға қатысы жоқ төбе деген сөзге анықтама бере кетелік. Мультидоғаға қатысы жоқ төбе деп, P предикатты төбесіне кіретін доғалар саны бірден артық болмайтын төбені айтамыз. P төбесін жою үшін мына ережені орындаймыз: Егер мультидоғаға қатысы жоқ P төбесі $g_1\ g_2\ ...\ g_n$ дизъюнктармен байланыста болса, онда оларға P пре-

дикатын қолдану нәтижесінде «таза» дизъюнкт алсақ, онда бұл Р төбесі тордан жойылады. *Төбелерді бөлу операторын қолдану.* Оператордың бұл түрі Р предикаттың төбесі үшін мультидоғалар бар жағдайында қолданылады. Тағы бір мысал қарайық. Дизъюнктердің мынандай жиыны бар деп есептейік:

1) P (x) $ → P (a);

2) → P(b);

3) P (a) →.

Тағы да бір дизъюнктердің предикаттар жиыны бар деп есептейік. Төбелерді бөлу операторын қолдана отырып, екі түрлі предикат төбесін алдық P, P_1. Олардың әрқайсысына бөлу операторын колдану аркылы ең аяғында «бос тор» деп аталатын шешімді аламыз.

Енді логикалық қорытынды арқылы жоспарлауда дәл емес түрдегі қорытынды табу әдісін қарастырамыз. *Дәл емес түрдегі қорытынды табу әдісі.* Біздер пайдаланатын мәліметтер мен білімдер көпшілік жағдайда дәл емес және толық болмауы мүмкін. Мәліметтер мен мағлұматтарда кездесетін осындай белгісіз жағдайлардың бірі – олардың дәл мағынаға ие бола алмау қасиетін атап өтелік. Өрнектелген ой-пікірдің қабылдайтын мағынасының дәл «шын» немесе «өтірік» дәрежесіне ие бола алмайтын қасиетін дәл емес қасиеті деп түсінеміз. Дәл емес түрдегі қорытынды табу қолданылатын үлгі түрлерінің бәрі болжамдылық ұғымына қатысты болғандықтан, мұндағы іздеу әдістерінің бәрі болжамдылық тұжырымдамасына байланысты қаралады. Дәл емес мәліметтер мен мағлұматтарды пайдаланатын үлгілер өзінің құрамында дәл еместік қасиетін өрнектейтін құрал мен оны іздестіру амалдарын ұстауы қажет. Дәл еместікті суреттейтін тіл табу үшін, бұл тілде дәл еместікті білдіретін құрылымдар болуы қажет.

«Қосылған» сипаттағы механизмдер. Бұл іс-әрекеттер дәл еместік өлшемнің шамасын анықтау – үнемі дәл өрнектерге арналған қорытындымен бірге жүріп отырады. Дәл еместік қорытынды жасауға арналған үлгі түрлері үшін дәл еместік өлшемін анықтайтын функциясы берілуі керек. Ол функцияның негізгі қызметі:

а) ереженің сол жағындағы антецедент (х)-тің дәл еместік өлшемін x_i – дердің өлшемдері арқылы табу, яғни $x = f(x_1, ..., x_n)$; x_i – осы антецедентке кіретін өрнектер;

ә) ереженің оң жағындағы консеквент – у-тің дәл еместік өлшемі арқылы анықтау: $y = h(r, x)$;

б) А – ой-пікірдің дәл еместік өлшемін, ережелер өлшемі арқылы анықтау. Ал бұл ережелерде олардың консеквенті болып табылады, А: $y^0 = g(y_1, ..., y_m)$.

Дәл еместік өлшемін анықтаудың шешім қабылдаудағы болжамдық әдістерге де тигізетін әсері бар. Бұл шешім дәлелденуге немесе алып тастауға тиісті кейбір болжамдарды біріктіріп қарауға мүмкіндік береді. Сарапшы жүйе программаларын құрғанда қолданылған болжамдық әдістерде Байес тәсілін қолданады. Олар *MYGIN, PROSPEKTOR* жүйелерін жасауға пайдаланылған. *PROSPEKTOR* жүйесінде өрнектелген ой-пікірлер өлшемі ретінде осыған қатысты оқиғалардың болжамдары берілген. Оқиғалар жиыны тор түрінде беріліп, олардың болжамдары есептеуге алынған. Жүйедегі ережелер мынадай түрде: «Егер Е шын болса, онда Н шын» және берілген торда әр {Е,Н} үшін бұл екі ереже бір уақытта болады не болмайды. Болжамдық функциясының өлшемін табуда дәл еместік логикасының белгілі формулалары қолданылады.

PROSPEKTOR және *MYGIN* жүйелеріндегі қорытынды шығару механизмдері қосылған сипатта болады және мұндағы дәл еместікті өлшейтін өлшемдер қарапайым скаляр түрінде беріледі. Бұл әдістердің әлсіз жағы, осында зерттелетін ой-пікірлер өрнегі бір-біріне тәуелсіз деген ойға негізделген. Ал өмірде, көп жағдайда олай бола бермейді. Бұл әдістер көбінде эвристикалық түрге жатады. Дәл еместік ой-пікірдегі өрнектерді шығаруға арналған механизмдер, егер осы өрнектер әдейі жасалған дәл еместік тілінде бейнеленген болса, соларға арналып жасалады. Бұл жағдайда қорытынды жасаудың әр қадамында дәл еместік өлшемді әрқашан қайтадан есептеп отыру қажеттігі туады [29].

Сонымен, біз *интеллектуалдық жүйе* деп аталатын программалардағы хабарларды іздеу іс барысының кейбір түрлеріне шолу жасап өттік. Бұл шолуды қысқаша тұжырымдап, мынаны айта аламыз: Білімді пайдаланып, өңдеп, үлгілеуде қолданылатын іздестіру әдістерінің барлығына қатысты бір кемшілігі бар. Ол – бұл тәсілдердің білімдер хабарларының құрылымдық түріне тәуелділігі. Классикалық үлгі бойынша жоспарлау әдістерінде бұл тәуелділік өте қатты сақталған. Бірақ адам баласы өзінің күнделікті іс-әрекетінде, жұмысында осы әдістердің барлық түрін араластырып қолдана береді. Логикалық қорытынды шығару арқылы жоспарлау әдістері үшін хабарлардың құрылымдық түрін өрнектейтін арнаулы тіл – *предикаттар тілі* ойлап табылған. Бұл тіл көмегімен адамның миындағы ойлау, іс барысына қатысатын ұғымдар мен нысандар олардың байланысын өрнектейтін формулалар жинағын құрап, оны қисынды ойлау жүйесі арқылы іс-әрекеттермен байланыстыруға болады. Білімдерді пайда-

лану үлгілеріне сарапшы жүйе программаларында кеңінен орын алып отырған – осы топтағы әдістер. Бұл әдістердің жасанды зерде программаларын құруда болашағы зор деп ойлаймыз.

3.3. Теоремаларды автоматты дәлелдеу әдістері

Бұл параграфта Предикаттарды есептеу мәселесін шешудегі тіл, Тілдің синтаксисі мен семантикасы, Предикаттарды есептеудің негіздері, Шығару ережелері мен үндестіру сияқты мәселелер қарастырылады [27].

Предикаттарды есептеу теоремасының негізгі ұғымдары. Автоматты түрдегі логикалық тұжырымдар жасау үшін, белгілі түрдегі формальды тіл қажет. Ол тілде ережелер құрып, логикалық тұжырымдар жасауға болады.

I-дәрежедегі предикаттарды есептеу. Осы аталған логикадағы жүйенің бір бөлігі математикадан тұрса, бір бөлігі сөйлесу табиғи тілден тұрады. Ой тұжырымдарын құру үшін, бұл тілді компьютерде пайдалануға болады. Кез келген тілде оның синтаксисі мен семантикасы болады. Синтаксисті анықтау үшін, тілдегі таңбалар алфавиті және осы таңбалардың бір-бірімен байланысу ережелері болады [24].

Синтаксис алфавит мынадан тұрады:

1) Пунктуация белгілері: , . ( ) .

2) Логикалық таңбалар: ~, => (~- белгісі «не» – «емес» деп; ал => «өз соңынан» – «влечет за собой» деп оқылады.

3) n – орындық функционалдық әріптер: f_i^n $(i \geq 1,\ n \geq 0)$ Олар f_i^o – константалы әріп деп аталды, және оларды a,в,c, деп белгілеп, ал f_i^n – f, g, h – деп белгілейді.

4) n – орындық предикаттық әріптер: P_i^n (i ≥ 1, n ≥ 0) (P_i^o – пропорционалды әріптер). Оларды P_i^n – оларды қарапайымдылық үшін P, Q, R – деп белгілеуді ұсынады.

Бұл таңбалар көмегімен әртүрлі өрнектер құруға болады. Өрнектер тобында мына терминдер анықталады:

1) *Термдер*

   а) әр константалы әріп – терм болып табылады.

   ә) егер – $t_1, t_2, ..., t_n$ (n≥1) – терм болса, онда f_i^n $(t_1, ..., t_n)$ терм болып табылады.

   б) одан басқа өрнектің ешқайсысы да терм емес.

2) *Атомдық формулалар.*

   а) пропорционалды әріптер атомдық формулалар болып табылады.

ә) егер $t_1, t_2, ..., t_n$ (n≥1) терім болса, онда $P_i^n(t_1, t_2, ..., t_n)$ – атомдық формула.

б) одан басқа өрнектің ешқайсысы да атомдық формула емес.

3) *ППФ – Дұрыс құрылған Формула.*

а) атомдық формула – ППФ

ә) егер А – ППФ болса, онда (~А) – ППФ.

б) егер А және В – ППФ болса, онда (A=>B) – ППФ болып табылады.

в) басқа өрнектің ешқайсысы ППФ емес.

Мысалы:

~P (a, g (a, в, a)) – ППФ.

P (a, в) => (~Q (c)),

(~P (a) =>P (в))) =>P (в),

~ P (a) => Q (f (a)).

ППФ-еместер:

~f (a),

f (P (a)) Q (f (a)) (P (в) => Q (c))).

Енді осы алфавитке мынаны қосайық: ∧ – «және», V – «немесе».

Мысалы: егер x_1 және x_2 кез келген ППФ болса, онда $x_1 \wedge x_2$, $x_1 \vee x_2$, – ППФ болады, яғни $x_1 \wedge x_2 = \sim (x_1 => \sim x_2)$, және $x_1 \vee x_2 = (\sim x_1) => x_2)$ – ППФ болады.

Семантика. ППФ-қа ішкі «мазмұн» беру үшін, оны белгілі бір аумаққа қатысы бар тұжырым ретінде тарату керек. Аумақ ретінде бүтін сандар жиынын немесе «15» ойынындағы конфигурациялар жиынын қарастыруға болады. Біз үшін қызықты болатын тұжырымдар осы аумақтағы элементтерді байланыстырып тұратын қарым-қатынастар арқылы іске асады. Мысалы, «Серік Талғаттың әкесі» деген тұжырымдағы аумақ – адамдар жиыны, ал қарым-қатынас – бинарлық «әкелік». Мысалы, **плюс** функциясы бүтін сан жұбын бүтін санға түрлендіреді. Мұнда қосу амалы орындалады. ППФ арқылы белгілі бір тұжырым жасау үшін, біз ППФ-ті бос емес D-аумағымен байланыстырамыз. Содан соң:

1) Әр константалық таңбаға (ППФ-тағы) D-дағы нақты бір элемент байланысады.
2) ППФ-дағы функциялық әріпке D аумағында нақты бір функция байланысады.
3) ППФ-дағы предикаттық әріпке D аумағындағы элементтер арасындағы нақты бір қарым-қатынас түрі байланысады.

Аумақты анықтау және аталған сәйкестіктерді ППФ-тің интерпретациясы немесе үлгісі деп аталады. Берілген ППФ-те әр атомдық формулаға T(true) немесе F(false) мәні беріледі. Бұл мәндерді беру, өте қарапайым өтеді. Егер предикатты әріптің термі D элементіне сәйкес келсе, онда T немесе F болады.

Мысалы: P(a,f(в,c)) – атом формуласы делік, және оның мынандай интерпретациясы бар: D- бүтін сандар жиыны: a – 2 саны, b – 4 саны, c – 6 саны, f – «+» қосу функциясы, P – «>» **үлкен** қатынасы (отношение). Мұндай интерпретацияда біздің атомдық формула былайша тұжырымдалады: «2 саны, 4 саны мен 6 санының қосындысынан үлкен». Бұл тұжырым қате болғандықтан, P(a, f(в, c)) – F қабылданады. Егер a = 11 болса, онда T болады. Әрине, басқа интерпретациялар да бар. Кейбір атомдық формулалар үшін олар T, ал кейбірі үшін F болуы мүмкін. ППФ-ті есептегенде мынандай ережелер пайдаланылады:

- егер x_1 – ППФ болса, онда $\sim x_1$ мәні T;
- егер $\sim x_1$ мәні F болса, онда x_1 – T болады;
- егер x_1 және x_2 екі кез келген ППФ болса, онда $(x_1 \wedge x_2)$ $(x_1 \vee x_2)$ және $(x_1 \Rightarrow x_2)$ мәндері шын болады.

Осындай есептеулерді, шындық мән беретін кестесі деп аталатын келесі кестеден анықтауға болады.

Шындық мәнін есептеу кестесі

x_1	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$	$x_1 \Rightarrow x_2$
T	T	T	T	T
F	T	F	T	T
T	F	F	T	F
F	F	F	F	T

Бұл есептеу әдісі – шын мән беруші кесте әдісі деп аталады. Егер кез келген ППФ-ның белгілі бір интерпретациясында осы әдіс бойынша есептеліп, онда T мәні алынса, онда аталған интерпретация көрсетілген ППФ-ті қанағаттандырады деуге болады.

Әдетте, кез келген аумақтағы элементтерге қатысы бар, белгілі бір тұжырымдар жасау қажет болғанда, осындай тұжырымды конъюнкция түрінде жазуға болады. Ол үлкен конъюнкцияларды көлемі үлкен өрнектерге қолданғанда, *әрқайсысы үшін* және *барлығына* деген сөздер қолданылады. Осы сөздерді математикалық логикада былай белгіленеді. ∀ (*әр біреуі үшін*). Мысалы, $(x_1 \wedge x_2 \wedge ... \wedge x_n)$ орнына $(\forall x)$ $P(x, f_i^o)$ деп жазамыз.

$\forall$ – (кері түскен А әрпі) символы таңбасы *жалпылық кванторы* деп аталады. Ал бұл таңбадан соң тұратын Х айнымалы жалпылық кванторына қатысы бар айнымалы деп аталады.

Осындай белгілеу (x_1 V x_2 V ... V x_n) – дизъюнкциялар үшін де бар. Дизъюнкция орнына $\exists$ (кері түскен Е әрпі) таңбасы пайдаланылады. Бұл квантор *өмір сүру, бар болу кванторы* деп аталады. Мысалы, мынандай тұжырым: «1 мен 100 арасындағы орналасқан кез келген бүтін жұп сандар үшін 1-ші сан екінші саннан үлкен». Бұл сөйлемді ППФ тілінде былай жазамыз:

$(\forall x)\ (\forall y)\ P(x, y)$ ППФ – F болады.

Мысалы, «кез келген бүтін сан үшін одан үлкен бүтін сан болады» деген тұжырымды былай жазуға болады:

$(\forall x)\ (\exists y)\ P(y, x)$.

Жалпылық және орындалу. Егер ППФ өзінің барлық түсіндірмелерінде Т (true) мәніне ие болса, онда ол жалпылық қасиетке ие. Мысалы: $P(a) \Rightarrow (P(a) \mid P(b))$.

Егер белгілі бір түсіндіруде ППФ-лар жиынындағы әр ППФ Т-мәніне ие болса, онда бұл түсіндіру осы жиынды қанағаттандырады деп есептеледі, яғни егер S-ті қанағаттандырған әр түсіндірме W-ны қанағаттандырса, онда ППФ-сы логикалық түрде S (ППФ)-тан шығады деп есептейміз. Бұл логикалық түрдегі шығару тұжырымдамасы осы предикаттарды есептеуді дәлелдеу негізіне салынған. Предикаттарды есептеудегі шешілмеу фактісі деп мынаны айтамыз: кез келген S-ППФ жиынында берілген W-ның S-тан логикалық түрде шығатынын көрсететін тиімді процедураның болмауы. Егер белгілі бір ППФ-лар жиыны ешқандай түсіндірмеде қанағаттанбаса, онда ол қанағаттанбаған (орындалмайтын) жиын деп аталады. Мысалы, W, S-те логикалық түрде шығарылса, онда SU {~W} қосылысы қанағаттанбайды және, керісінше, егер SU {~W} қанағаттанбаса, онда W S-тен логикалық түрде шығарылады. Осы түйінделген нәтижені барлық дәлелдеуді қажет ететін есептерге біркелкі түр, форма беру үшін пайдалануға болады. Яғни W-ның S-тан логикалық түрде шығатынын дәлелдеу үшін, SU {~W} қанағаттанбайтынын жалпылық кванторының бар болу кванторымен қатар орналасуға кез келген Х – үшін Y-тің бар екенін білдіреді.

Жалпылық мәнде болуы мен орындалуын дәлелдесе жеткілікті. Бұл мәселе күрделі болғанымен, оны шешуге мүмкіндік беретін тиімді процедуралар бар. Ол үшін аталған жиындағы ППФ-ды арнайы бір түрге келтіру қажет. Ол түр сөйлемдер (clouse) деп аталады. Есептеу

предикаттарындағы кез келген ППФ-ті сөйлемдер түріне келтіруге болады. Ол үшін оған қарапайым бірінен соң бірі орындалатын операциялар қолданылады. Мысалы, мынандай ППФ берілген делік:

$(\forall x)\ \{P(x) \Rightarrow \{(\forall y)\{P(y) \Rightarrow P(f(x, y))\}\ \wedge \sim (\forall y)\ \{Q(x, y) \Rightarrow P(y)\}\}\}.$

оған қарапайым операциялар қолдану арқылы мынандай сөйлемдер аламыз:

$\sim P(x)\ \mathrm{V} \sim P(y)\ \mathrm{V}\ P(f(x, y));$

$\sim P(x)\ \mathrm{V}\ Q(x, g(x));$

$\sim P(x)\ \mathrm{V} \sim P(g(x)).$

Сөйлемдер дизъюнкциялар (V белгісі) мен терістеу (~ белгісі) таңбалармен біріккен ППФ-ті құрайды. Осы алынған сөйлемдер жиыны қанағаттанбаған болса, онда жоғарыда берілген ППФ осы жиыннан логикалық түрде шығарылған деп есептейміз. Енді осы тұжырымдарды мысал арқылы қарастырамыз. Мысалы, *«Егер Арман үнемі Васямен бірге жүрсе, ал Вася мектепте болса, онда Арман қайда?»*.

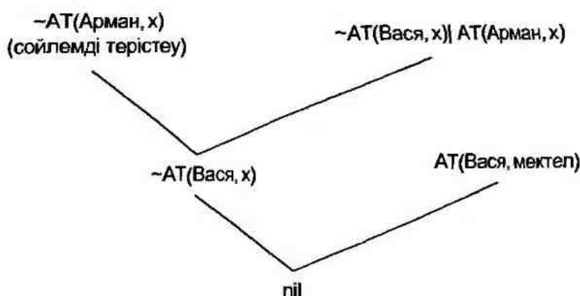
Бұл мысалда *«Арман қайда?»* деген сұраққа жауап беру үшін, ППФ S-тен шығатынын көрсетсек жеткілікті. Мұнда екі факті бар [27].

1) кез келген X үшін AT ( ) – бір жерде болу предикаты.

2) AT (Вася, мектеп) – Вася мектепте предикаты.

(∃x) AT (Арман, X) – формуласы Арманның кез келген X деген орында бола алатынын білдіреді. (Мысалдың терістеу ағашы 3.1-суретте келтірілген).

«Арман қайда?» деген сұраққа жауап табу үшін, алдыңғы ППФ-тың S-тен шығарылатынын дәлелдесек жеткілікті. Шешудің негізгі ой арқауы мынада: бар болу кванторын бойында жинақтаған ППФ-ын түрлендіру, соның нәтижесінде бар болу кванторына жататын және жауап бола алатын айнымалыны табу. Егер берілген фактілер арқылы жауап табылса, онда ППФ S-тен шығарылады деген қорытындыға келеміз.



3.1-сурет. Мысалдың терістеу ағашы

Дәлелдеуі:

1) Ең алдымен дәлелдеуді қажет ететін ППФ терістеледі.
2) Осы терістеу S жиынына қосылады.
3) Кеңейтілген жиын мүшелері сөйлем түріне келтіріледі.
4) Резольвенция принципі көмегімен сөйлемдер жиынының қанағаттанбайтындығы көрсетіледі.

Дәлелдеуге қажетті ППФ жорамал деп аталады, ППФ-тен алынатын сөйлемдер *аксиомалар* деп аталады. *«Арман қайда?»* – деген сұраққа жауапты терістеу ағашынан табу үшін, мыналарды орындаймыз:

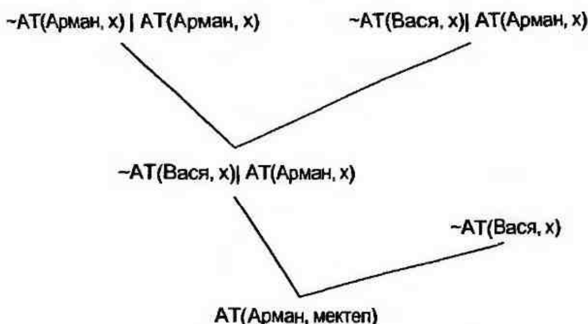
1) Әр сөйлемге оның терістеуін қосамыз, яғни (WV ~ W) – тавтологиясын орындаймыз.

2) Терістеу ағашының құрылымына байланысты ағаш түбірінде белгілі бір сөйлем пайда болғанға дейін резольвенциялар орындалады.

Екі аталық сөйлемнен пайда болған сөйлемді – ***Резольвента*** деп атаймыз. Сонымен қатар бұл екі сөйлем белгілі бір жиынның түсіндірмесіне жатуы керек. Мысалы, *~P(f(y))VQ(f(y))* және *~Q(f(y))* деген екі сөйлемнен мына сөйлемді *~P(f(y))* шығаруға болады. Бұл шығарылған сөйлем аталған екі сөйлемнің резольвентасы болады.

Резольвенция принципі дегеніміз – екі аталық сөйлемнен резольвентаның пайда болу үдерісін айтамыз. Резольвенцияны пайдаланатын терістеу үдерісі граф тәрізді құрылымдарда өтеді. Бұл құрылымдардың әр төбесінде бір сөйлем жазылады. Егер соңғы екі төбедегі екі сөйлем өзара шешілсе, онда олардың резольвентасы олардан кейінгі болатын төбеге жазылады. Бұл пайда болған төбе алдыңғыларымен қабырғалары арқылы байланыста болады. Осылайша пайда болған терістеу графының түбірі бос сөйлем болып табылады. Әдетте, оны nil таңбасымен белгілейді.

$\exists x[\sim AT(Арман, x)]$

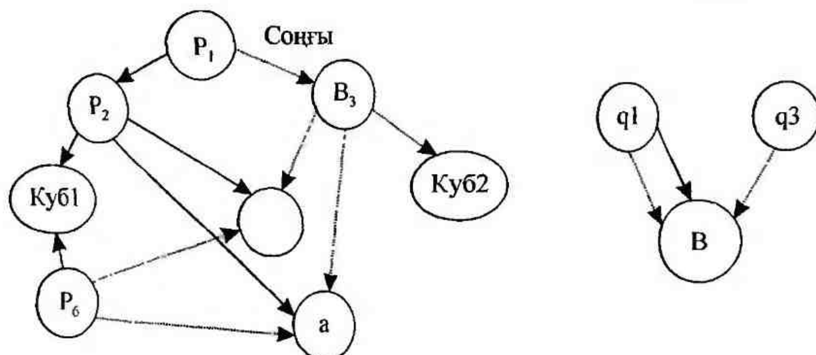


3.2-сурет. Мысалдың жауап-сөйлемі бар графы

Ал түрленген граф түбірінде алынған жауап-сөйлем болады. Жоғарыда келтірілген мысал түрінің жауабы 3.2-суреттегідей болып келеді. Сонымен, резольвенция принципін қолдану нәтижесінде адам жететін жауапты бере алатын шешім түрі алынады.

3.4. Предикаттарды есептеудегі іздеу әдістері

Кеңістіктегі күйі бойынша іздеу әдісі. Сыртқы дүниенің жалпыланған жағдайлары ішінен оның ең алғашқы күйі деп аталатын (ол әдетте біреу), ең соңғы күйі деп аталатын жағдайды бөліп қарауға болады. Бұндайда іс-әрекетті жоспарлау мақсаты – алғашқы күйден нысанаға дейінгі жеткізетін жолды табу. Есептерді кеңістіктегі күйі бойынша жоспарлағанда, оның мынандай жағдайлары бейнеленуі қажет: хал-жайы, операторлар жиыны, сол операторлардың хал-жайға ететін әсері [27].



3.3-сурет. Күй жағдай кеңістігіндегі граф төбелері

Граф төбелері күй-жайды, ал оларды қосып тұрған сызықтар, доғалар операторларды білдіреді. Егер кез келген доға n_i төбесінен ni төбесіне қарай бағытталатын болса, онда nj еншілес төбе, ал ni аталық төбе деп аталады.

Егер $n_{i1}, \ldots, n_{jk}$ төбелер тізбегі болса және n_{ij} төбесі n_{ij-1} төбесі үшін еншілес төбе болса, j = 2, k, онда k-жолы деп n_{i1} төбесі мен n_{ik} төбесін қосып тұрған сызықтар жиынын айтамыз. (3.3-суретті қара) ол жол бағытталған сызықтармен берілген. Сонымен, <A, B> түрінде берілген есептің шығару жолын табу, A-төбесінен B-төбесіне дейінгі жолды табу мәселесі болып табылады [27].

Есепті кеңістіктегі күйі бойынша құру. Күйлерді сипаттау. Күйлер кеңістігін пайдаланып есепті сипаттау үшін, біз ең алдымен бұл есептегі ***күй*** деген не соны анықтап алуымыз қажет. Осы күйлер кеңістігіндегі күйлердің сипатын анықтау үшін оны қандай ***қалыппен (форма)*** сипаттаймыз, соны анықтау мәселедегі маңызды кезеңдер қатарына жатады. Жалпы жағдайда күйлерді сипаттауға айнымалыларды анықтайтын кез келген құрылымдар жарай береді. Бұндай құрылымдарға: таңбалар қатары, векторлар, екі өлшемді массивтер, ағаштар, тізімдер сияқты қалыптарды жатқызуға болады. Әдетте, таңдап алынған сипаттау қалыбы шешілуге тиісті есептің кейбір физикалық қасиеттеріне де байланысты болады. Мысалы, көпшілікке таныс «15» ойынының күйлерін сипаттау қалыбы есебінде 4x4 өлшемді массивті таңдау ойын табиғатына да сай болар еді. Күйлерді сипаттайтын қалыпты таңдағанда, оны басқа күйге түрлендіретін оператордың жеңіл, әрі оңай болуына да көңіл аудару қажет.

Операторлар. Күйлер және операторлар. Есептегі іздеу әдістерін талдаудан бұрын осы есепке қатысты күй және оператор ұғымдарын кеңінен зерттеп алу қажет. Жоғарыда аталған «15» ойыны үшін есептің күйі болып белгілі бір қалыптағы фишкалардың орналасу реті табылады. Ойынның бастапқы және мақсатты конфигурациялары есептің бастапқы және мақсатты күйлері болып табылады. Есептегі *күйлер кеңістігі* деп бастапқы күйден мақсатты күй аралығындағы фишкалардың мүмкін болатын орнын ауыстыру нәтижесінде туатын барлық күйлер жиынтығын айтамыз. Көптеген есептердің өте үлкен (кейбіреуі шексіз) күйлер кеңістігі бар. Есептің *операторы* бір күйді екінші түрге түрлендіреді. Мысалы, «15» ойыны үшін фишкаларды қозғайтын төрт оператор бар: бос клетканы (ол жұлдызша, боялған, т.б. болуы мүмкін) *солға, оңға, жоғары, төмен* жылжыту. Кей кезде таңдап алынған оператор белгілі бір күйге қолданыла алмайтын жағдайда да болуы мүмкін. Күйлер мен операторлар тілінде белгілі бір проблеманы шешу деп бастапқы күйге оны мақсатты күйге түрлендіретін операторлар тізбегін қолдануды айтамыз. Әдетте, күйлер кеңістігін граф түрінде бейнелеу ыңғайлы. Оның бастапқы төбесінде, әрине, бастапқы күй болады да, төбелерді қосып тұрған сызықтар-доғалар қолданылатын операторға сәйкес келеді. Қолданылатын операторлар бір күйден екінші күйді тудырады. Сондықтан оларды күйлер жиындарында анықталған функциялар есебінде қарауға болады. Функциялар өз мәндерін осы жиыннан анықтап отырады. Біздің қарастырып отырған есебімізді шешу үдерісі күйлер сипаттамасымен жұмыс

істеуге негізделгендіктен, біз операторларды осы сипаттаулардың функциясы деп, ал олардың мәнін жаңа туындаған күйлер деп қабылдаймыз. Жалпы жағдайда операторлар деп бір күйлерді екінші күйлерге түрлендіретін *есептеулерді* қарастыруымызға да болады.

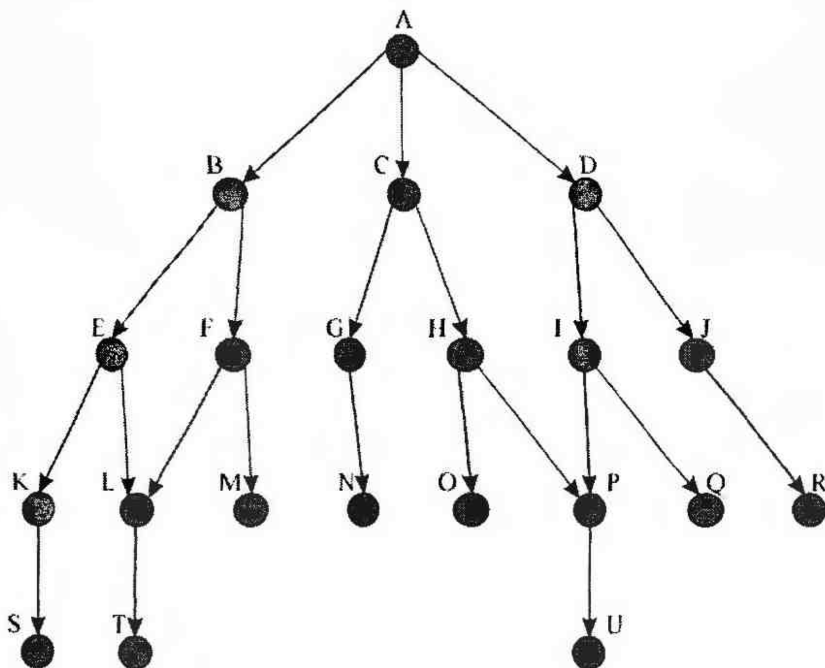
Негізгі ұғымдар мен терминдер. Сонымен, белгілі бір есепті күйлер кеңістігінде бейнелеу үшін, мына ұғымдарды анықтап, белгілеу қажет:

а) күйлер сипатының қалыбын, әсіресе бастапқы күй қалыбын анықтау;

б) операторлар жиыны мен олардың сипатталған күйлерге ететін әсерін анықтау;

в) мақсатты күй сипатының қасиетін анықтау.

Күйлер кеңістігін бағытталған граф есебінде бейнелеу ыңғайлы. *Граф түріндегі жазба.* Граф *төбелер* (шектелген болуы міндетті емес) жиынынан тұрады. Графтағы кейбір төбелер жұбы *доғалармен* байланысқан және олар бір төбеден екіншісіне қарай бағытталған. Мұндай графтар *бағытталған графтар* деп аталады (3.4-суретін қара).



3.4-сурет. Кеңістіктегі күйі бойынша іздеу графы

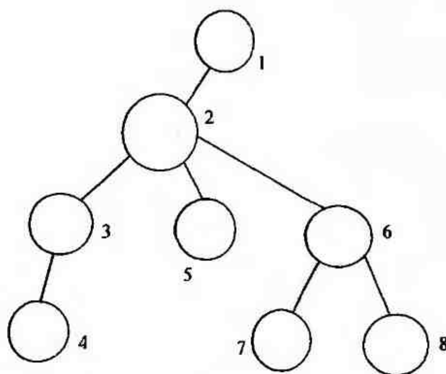
Егер кейбір доға n_i төбесінен n_j төбесіне қарай бағытталған болса, онда n_j төбесі n_i төбесі үшін еншілес төбе, ал n_i төбесі n_j төбесі үшін аталық төбе деп аталады. Егер екі төбе бір-бірі үшін еншілес төбелер болса, онда бағытталған доғалар жұбы граф қабырғасы деп аталады. Граф құрылымы кеңістіктегі күйлерге қолданылғанда, оның төбелерінде күйлер, ал оның доғаларында операторлар сипатталды.

3.4.1. Күйлер кеңістігіндегі іздеу әдістері

Жоғарыда аталып өткендей, күйлер кеңістігіндегі іздеу әдістеріне *соқыр әдістер* тобы жатады. Бұл әдістің екі түрі болады: тереңнен іздеу, жайылып іздеу.

Тереңнен іздеуде әрбір күй аяғына дейін тексеріледі де, басқа күйлердің әсері еске алынбайды. Осы әдістегі күйлерді іздеу жолдары 3.5-суретте сандар ретімен цифрланған. Тереңнен іздеу жолын өте биік болып келетін ағаштар үшін қолдану тиімсіз. Өйткені керек төбенің қасынан ұзап кетеді де, басқалардың бәрінен өту үшін көп уақыт кетеді.

Күйлер кеңістігіндегі осы іздеу әдістерінің бірі – тереңнен іздеу тәсілін қарастырайық. *Тереңнен іздеу (Depth-first search, **DFS**).* Бұл әдісте ең алдымен ең соңғы құрылған төбелер ашылады.

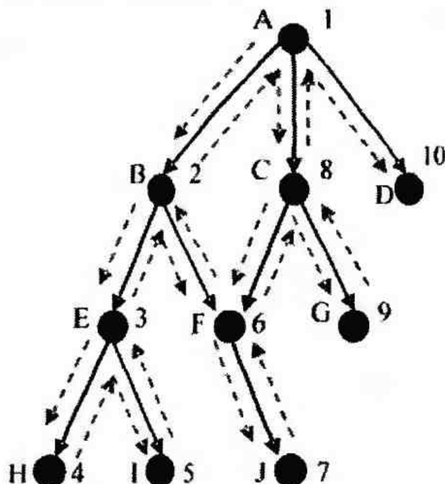


3.5-сурет. Тереңнен іздеу әдісі

Төбелерді ашу үдерісі деп осы төбеге операторлар жиынын қолдануды айтамыз. Таңдап алынатын төбелер тереңдігі былайша анықталады: ағаш түбірінің тереңдігі нөлге тең, ал келесі төбенің тереңдігі оның алдындағы төбе тереңдігіне бірді қосу арқылы

анықталады. Сонымен, іздеу ағашындағы ең терең төбе есебінде осы сәтте ашылуға тиісті төбе алынады. Бұл жол бітпейтін шексіздікке апаруы мүмкін болғандықтан, қайтару жолдарын да анықтап отыру қажет. Ол үшін біз алдын ала белгіленген тереңдік мөлшерін береміз. Осы тереңдіктен кейін ашылатын төбелерде мақсатты күй болмаса, онда біз белгілі бір тәсілдерге сүйене отырып, қаралмаған төбелерге қайтып келеміз. Тереңнен іздеу әдісінде 3.4-суретінде көрсетілгендей, төбелер мына тәртіппен ашылады: A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R.

Күйлер кеңістігі графында іздеу алгоритмдерінің біріне ***бектрекинг*** (backtrack) деп аталатын қайтаруы бар іздеу тәсілін атауға болады [1, 2, 39]. Бұл тәсіл есептің мүмкін болатын шешімдері көп болатын жағдайда қолданылады. Бектрекинг тәсілінің мәні мынада: есептің іздеуінде тармақталатын жолдарда, оның біреуімен кетіп, онда мақсатты күй болмаса, қалған жолдарға қайтып келу үшін осы тармақты еске сақтап қою. Жалпы жағдайда мұндай қайту тармақтары көп болуы да мүмкін. Оларды *бектрекинг нүктелері* немесе *тармақтар* деп атайды. Пролог тілінің кейбір нұсқаларында [5] және Плэнер тілінде [2] осы бектрекинг тәсілін іске асыратын арнайы механизмдер бар. Ол – бектрекинг нүктелерін еске сақтау мен осы нүктелерге қайтып келуді тілдің интерпретаторы автоматты түрде іске асырады деген сөз. Программалаушы тек бектрекинг нүктелерін анықтау мен қайту үдерісін басқаруды іске асыратын амалдарды орындауы қажет.



3.6-сурет. Қайталап іздеу графы

Осы тәсілді бейнелеу үшін, 3.6-суреттегі графты қарастырайық. Үзік сызықтар іздеу бағытын көрсетеді. Әр төбенің қасындағы сандар оларды ашу тәртібін білдіреді.

Іздеу алгоритмі келесі түрде болады:

1) Зерттелмеген күйлер тізімі қалыптасады (***NSL***), ол осы күйлерге қайта айналып келу үшін қажет.
2) Сәтсіз күйлер тізімі болады (***DE***), ол алгоритмді қажетсіз іздеу жолдарынан сақтау үшін керек.
3) Ағымдағы жолдың түйіндер тізімі (***SL***), ол мақсатты күйге жеткендегі жолдың тәртібін көрсетеді.
4) Әр жаңа күй осы аталған тізімдерге кіруі қадағаланып отырады, ол есептің шексіз кезеңге кетіп қалуынан қорғап отырады.

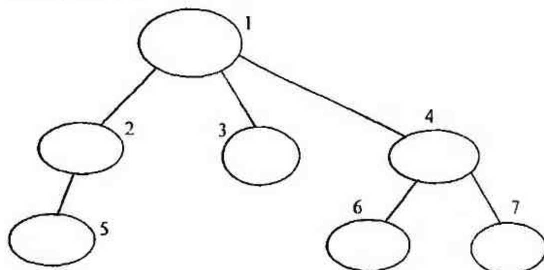
Жалпы, тереңнен және жайылып іздеу алгоритмдерінің барлығы да қайтаруы бар іздеу жолын пайдаланады. Олар кеңістіктегі күйлердің жылжуын қамтамасыз ететін ***open*** және ***closed*** деп аталатын тізімдерді пайдаланады. Мұндағы ***open*** тізімі (ол ***NSL*** тізіміне ұқсас) тұқымдары әлі зерттелмеген күйлерді тудырады. Күйлерді тізімнен алып тастау тәртібін іздеу әдісі анықтайды. Ал ***closed*** тізіміне (ол ***DE*** және ***SL*** тізімдеріне ұқсас) зерттелген күйлер енгізіледі. Тереңнен іздеу әдісіне жататын осы ***open*** және ***closed*** тізімдері 3.1-кестесінде келтірілген. Кестеде 3.4-суретіндегі күйлер сипатталған.

3.1-кесте

DFS тәсіліндегі күйлердің **open** және **closed** тізімдері

Итерациялар	***open*** тізімі	***closed*** тізімі
1	[A]	[]
2	[BCD]	[A]
3	[EFCD]	[BA]
4	[KLFCD]	[EBA]
5	[SLFCD]	[KEBA]
6	[LFCD]	[SKEBA]
7	[TFCD]	[LSKEBA]
8	[FCD]	[TLSKEBA]
9	[MCD]	[FTLSKEBA]
10	[CD]	[MFTLSKEBA]
11	[GHD]	[CMFTLSKEBA]

Тереңнен іздеу әдісіне жататын іске асыру механизмі немесе тізімдерді өңдеу тәсілі ретінде программалау алгоритмдерінде «стек» механизмі таңдалып алынады. Бұл іздеу әдісінде ***open*** тізімі үшін дүкен типіндегі стек механизмі орындалады, оны әдебиетте ***LIFO*** «last-in-first-out», яғни «соңғы келді – бірінші қызмет көрсетілді» құрылымы деп атайды. Бұл механизмде күйлер тізімнің сол жағынан қосылып, сол жағынан өшіріледі.



3.7-сурет. Жайылып іздеу әдісі

Жайылып іздеуде барлық күйлер ең төменгі жағында орналасқан болса, бұл әдіс жолын қолдану тағы да тиімсіз болады. Аталған әдістегі ашылатын төбелер 3.7-суретте рет-ретімен цифрланған.

Толық қарау әдісі. Жайылып іздеу. (***BFS***, Breadth-first search). Жайылып іздеуде төбелер қалай құрылса, солай ашылады. Бұл әдісте мақсатты күй бар болса, ол міндетті түрде табылады. Егер ондай жол жоқ болса, онда іздеу сәтсіз деп хабарланады. Бұл жағдай іздеу графының шегі болғанда орындалады, ал егер граф шексіз болса, алгоритм өз жұмысын ешқашан бітірмей тұрып алады. Қорыта айтсақ, бұл әдісте ағымдағы төбеге байланысты пайда болған барлық төбелер қаралып өтіледі. Келесі төбені таңдау принципі – оның алдындағы төбе қарастырылады. Жоғарыда 3.4-суретте көрсетілген графтағы күйлер жайылып іздеуде мына тәртіппен зерттеледі: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U. Келесі 3.2-кестеде жайылып іздеу әдісіне қатысты ***open*** және ***closed*** тізімдері келтірілген.

Жайылып іздеу әдісіне жататын іске асыру механизмі немесе тізімдерді өңдеу тәсілі ретінде программалау алгоритмдерінде «кезек» механизмі таңдалып алынады. Бұл іздеу әдісінде ***open*** тізімі үшін кезек механизмі орындалады, оны әдебиетте ***FIFO*** «first-in-first-out», яғни «бірінші келді – бірінші қызмет көрсетілді» құрылымы деп атайды. Бұл механизмде күйлер тізімнің оң жағынан қосылып, сол жағынан өшіріледі.

3.2-кесте

BFS тәсіліндегі күйлердің **open** және **closed** тізімдері

Итерациялар	***open*** тізімі	***closed*** тізімі
1	[A]	[]
2	[BCD]	[A]
3	[CDEF]	[BA]
4	[DEFGH]	[CBA]
5	[EFGHIJ]	[DCBA]
6	[FGHIJKL]	[EDCBA]
7	[GHIJKLM]	[FEDCBA]
8	[HIJKLMN]	[GFEDCBA]
9	[JKLMNOPQ]	[HGFEDCBA]
10	[KLMNOPQR]	[IHGFEDCBA]
11	[NOPQRSTU]	[JIHGFEDCBA]

Жайылып іздеу былайша орындалады: іздеудің бастапқы деңгейіне 0 таңбасы беріліп, одан кейінгі төбелерді іздеу 1 деңгей одан кейінгі 2 т.б. таңбаларында өтеді. Осылайша мақсатты күйге жеткенше деңгейлердің әрқайсысы өз таңбасымен өтіп, ондағы төбелердің ашылу үдерістері орындалады. Жайылып іздеуде барлық күйлер ең төменгі жағында орналасқан болса, бұл әдіс жолын қолдану тағы да тиімсіз болады.

Бұл әдістің екі жолы да көп уақытты керек ететіндіктен, бағытталған әдістерді қолдану қажеттігі бар.

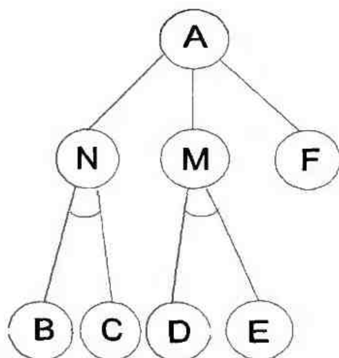
Эвристикалық әдістер. Бұл тәсілдер тобына алдыңғы екі тәсілден басқа жолдардың барлығы кіреді. Яғни кеңістіктегі күйлерін белгілі бір қалыптармен өрнектеп, мақсатты күйді іздеуде қолданылатын кез келген алгоритм жолдарын осы топқа жатқызуға болады. Мысалы, осындай тәсілдің бірі – *тең бағалар әдісі.* Бұл тәсіл бойынша бастапқы төбеден мақсатты төбеге дейін бағасы ең арзан болатын төбе арқылы өтетін жол таңдалып алынады.

3.4.2. Кеңістіктегі есептер жағдайы бойынша есептеу

Бұл жоспарлау берілген есепті ең қарапайым түрге келтіргенге дейін бөлшектеп, карауға болады деген ойға негізделген. Осы бөлшектелген

әрбір есептің шешімдер жиыны берілген есептің шешімі болып табылады. Есепті осындай бөлшектеу түрі 3.8-суретте көрсетілген [24]. Оны ЖӘНЕ/НЕМЕСЕ графы деп атаймыз. Мұндай графтағы кез келген төбе бір-бірімен байланған балалық төбелермен (ЖӘНЕ төбелер), не басқа түрдегі төбелермен (НЕМЕСЕ төбелер) байланыста болады. Мысалы, суреттегі A алғашқы төбе де, ол N, M, F төбелерімен НЕМЕСЕ болады да, ал N төбесі B мен C төбелерімен, ал M төбесі D, E төбелерімен ЖӘНЕ байланысында болады.

Шешімді іздеуде кеңістіктегі есептер жағдайы бойынша жоспарлау көбінде жақсы нәтиже береді. Алайда бұл әдісті есепті шешуге кірісердегі жалпы жағдайға қолданған тиімді, ал ары қарай бөлшектенген есеп бөліктеріне кеңістіктегі күйі бойынша жоспарлауды қолданып, есепті бөлшектеу әдісінің бір жолы деп те қарауға болады.

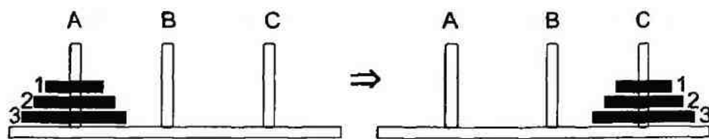


3.8-сурет. ЖӘНЕ/НЕМЕСЕ графы

Компьютерлік ойын программаларын шешуге қолданылатын кеңінен тараған бұл тәсілдің кеңістіктегі күйі бойынша іздеу әдістерінен ерекшелігі – бұл әдістің қолдану аумағы алдыңғыға қарағанда ауқымды әрі кең болып келеді. Яғни үлкен есепті кіші есептерге бөлшектеу (ҮЕКЕБ) немесе оның әдебиетте қалыптасқан басқаша атауы Редукция әдісі өз құрамында кеңістіктегі күйі бойынша іздеу жолдарын сақтап қолданады. Бұл әдісте мәселе шешілуге тиісті есепті үлкен есеп деп қарап, оны көптеген кіші есептерге бөлшектеу арқылы тиімді шешім жолына жетуді қарастырады. Әрбір кіші есеп алдыңғыға қарағанда қарапайым болып келеді, оны белгілі бір әдістермен, мысалы, жоғарыдағы соқыр әдістердің бірімен шешуге болады. Редукция әдісі есепті өз тәсілдерімен де шеше алады. Ол үшін есепті бөлшектеу

үдерісін ары қарай жалғастырып, алынған кіші есептерден тағы одан кіші есептерді алып, ең соңында *қарапайым* есеп болғанға дейін бөле береміз. Ал *қарапайым есеп* дегеніміз – шешімі белгілі есеп. Мәселені қоюдың мұндай жолын есепті кіші есептерге келтіру немесе *есептің редукциясы* деп атайды.

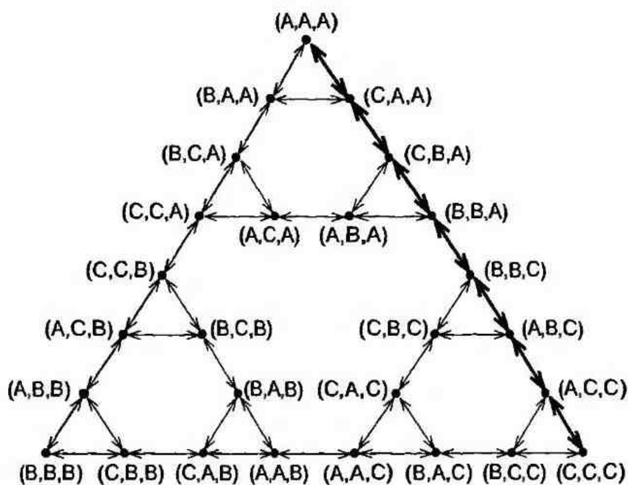
Редукция мәнін толық түсіну үшін, көпшілікке кеңінен таныс классикалық есеп болып кеткен Ханой мұнарасы немесе оны пирамида жайындағы есеп деп те атайды, есебін қарастырайық [24]. Есеп шарты бойынша үш қазық берілген. Оны А, В, С әріптерімен белгілейік. Сонымен бірге көптеген диаметрі әртүрлі ортасы тесік дискілер берілген. Олар қазықтарға бірінің үстіне бірі кигізілген. Дискілер санын қарапайымдылық үшін 3-ке тең деп алдық. (Біз осы есепке қатысты суреттер мен алгоритм түрлерін [39] әдебиеттегі авторлар нұсқасы бойынша алдық). Бастапқы кезде барлық дискілер А қазығында орналасады және есептің шарты бойынша кіші дискілер үлкендердің үстінде болады, керісінше орналасуына рұқсат жоқ (3.9-суретті қара). Есептің мақсаты барлық дискілерді С қазығына орналастыру қажет. Орындалуы тиісті ережелер: жылжытуды тек үстіңгі дискіден бастайды, кіші дискінің үстіне үлкен дискіні қоюға болмайды. Келесі 3.7-суретте осы Ханой мұнарасы есебінің бастапқы және мақсатты күйлері қатар көрсетілген.



3.9-сурет. Ханой мұнарасы есебі

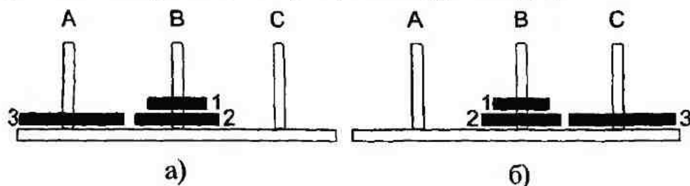
Бұл есепті кеңістіктегі күйлері бойынша сипаттап, оның күйлері мен қолданылатын операторларды қалыптастырсақ, біз, әрине, соқыр әдістердің бірімен оны шешуімізге болар еді. Мұндағы күйлер үш элементтен тұратын тізіммен беріледі. Тізімдегі әр элемент дискілер ретін көрсетеді. Мысалы, бастапқы күй (ААА) тізімімен берілсе, ал мақсатты күй (ССС) тізімімен анықталады. Мұндағы (ААА) тізімі барлық үш дискінің бірінші А қазығында тұрғанын көрсетсе, (ССС) тізімі барлық үш дискінің С қазығында екенін көрсетеді.

Осы пирамида жайындағы есептің күйлерінің толық кеңістігі 27 төбесі бар графтан тұрады (3.10 суретті қара). Суреттегі қою түсті бағытталған доғалар есептің ең қысқа шешім жолын көрсетеді, ол дискілерді 7 рет жылжытудан тұрады.



3.10-сурет. Ханой мұнарасы есебіндегі күйлер кеңістігі

Бұл шешімді дискілердің барлық жылжытуларын орындай отырып та алуға болады. Дегенмен біз қарастырғалы отырған редукция әдісі оған жетудегі қысқа жолды ұсынады. Редукцияның негізгі ой желісі мынада: алдымен ең төменгі үшінші дискіні үшінші қазыққа (С) орналастыру қажет, ол үшін үстіңгі екі 1 және 2 диск В қазығына өз реттерімен келіп орналасуы керек (3.11 а суретті қара).



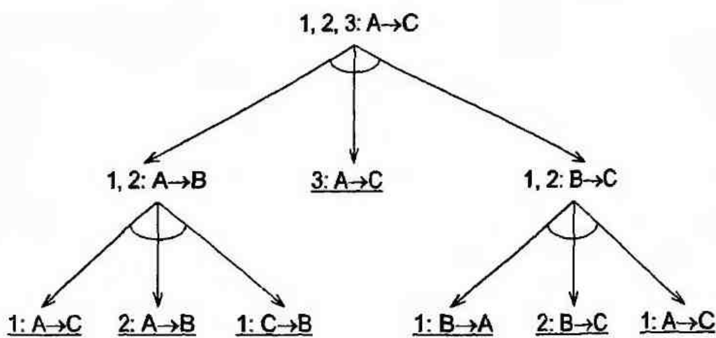
3.11-сурет. Ханой мұнарасы: екі шешілуші күйлер

Сонымен, бастапқы есепті мынандай үш кіші есепке бөлуге болады:

1) А қазығындағы 1 және 2 дискіні В қазығына орналастыру (1, 2 : А→В);
2) А қазығындағы 3 дискіні С қазығына орналастыру (3 : А→С);
3) В қазығындағы 1 және 2 дискіні С қазығына орналастыру (1, 2 : В→С).

Осы күйлер көрінісін 3.9,б-суретінен көруге болады. Онда үшінші есептің күйлері көрсетілген.

Әр кіші есеп алдыңғы есепке қарағанда қарапайым. Шынында, бірінші және үшінші есепте небары екі дискіні жылжыту қажет болса, екінші есепті *қарапайым* деп атауға болады, онда тек бір ғана 3 диск С қазығына жылжытылады. Бірінші және үшінші есептерге тағы да редукция әдісін қолданып, олардан *қарапайым* есептерді шығаруға болады. Редукцияда өтетін барлық үдерісті сызбанұсқа түрінде граф ағашы ретінде бейнелеуге болады (3.12-суретін қара). Ағаш төбелері шешілуге тиісті есеп/кіші есептерге сәйкес келсе, ал ағаш жапырақтары дискілерді жылжыту есептеріне сай болады. Ал граф төбелерінің доғалары редукцияланатын есептерді олардың кіші есептерімен байланыстыратын сызықтарды білдіреді.



3.12-сурет. Ханой мұнарасы есебінің редукциясы

Сонымен, редукция әдісінде де біз кеңістік аламыз, бірақ ол онда күйлер емес, есептер мен кіші есептер болады (яғни олардың сипаттамалары). Операторлар есебінде осы үлкен есепті кіші есептерге келтіретін операторлар болады. Анығырақ айтсақ, *редукцияның әр операторы* үлкен есеп сипатын кіші есептер сипатына түрлендіреді. Бұл алынған кіші есептер жиынында мақсатқа жететін шешім түрі болады [2].

Редукция әдісінде де, кеңістіктегі күйлер әдістеріндегідей барлық мүмкін болатын жолдарды қарау мәселесі де туады. Шынында, редукцияның әр кезеңінде қолдануға болатын бірнеше оператор болуы мүмкін (яғни есепті кіші есептерге бөлшектейтін бірнеше тәсіл), демек, кіші есептер жиындарының бірнеше түрі болуы мүмкін. Бұл жағдайда редукция үдерісі бастапқы есеп қарапайым есептерге бөлшектенгенге дейін жүре береді. Ал қарапайым есеп шешімі белгілі.

Негізгі ұғымдар мен терминдер. Кеңістіктегі күйлері бойынша шешілуге тиісті есептер сияқты редукция әдісіне негізделген есептерде де белгілі бір қалыптарды анықтап алу қажет:

- есеп/кіші есеп сипаты мен бастапқы есеп сипатының қалыптары;
- операторлар жиыны мен олардың есептің сипатына ететін әсері;
- қарапайым есептер жиыны.

Осы аталған құрылымдар есеп шешуін табуға іздеуді ұйымдастыруға қажетті *есептер кеңістігін* білдіреді. Ал есеп/кіші есеп сипаттарының қалыптары жайында сөз қозғасақ, онда оларды кеңістіктегі күй тәсілдеріндегідей бастапқы күйлер, операторлар, мақсатты күйлер мен олардың қасиеттері деген сияқты сипаттап келтіруге болады. Бұл жағдайда қарапайым есептер есебінде күйлер кеңістігінде бір қадамда орындалатын есептерді алуға болады.

Тағы бір атап өтетін жағдай – кеңістіктегі күйлері бойынша іздеу тәсілдерін редукция әдісінің бір бөлігі есебінде қабылдау. Яғни редукция жағдайындағы операторды қолдану деген бастапқы есепті қарапайым есептерге келтіру болып табылатындықтан, бастапқы күйді мақсатты күйге дейін редукциялайды деп айта аламыз. Және осы жағдайда кіші есептер жиындарында тек бір ғана элемент болады, яғни біз редукцияланатын есептің оған ұқсас қарапайым есепке ауыстырылатынына куә боламыз.

ЖӘНЕ/НЕМЕСЕ графы. Шешуші граф. Редукция есебінің кіші есептерге бөліну үдерісін әдетте, графқа ұқсас құрылымдармен өрнектейді. Олардың төбелерінде есептер мен кіші есептер, ал оларды байланыстырып тұратын доғалар редукцияланатын есеп пен одан туындайтын кіші есепті жұп төбелермен қосып тұрады. Доғалардың бағыты редукция бағытын білдіреді. Мұндай құрылымдар *ЖӘНЕ/НЕМЕСЕ графтары* деп аталады (3.8 және 3.12-суреттерін қара).

Осындай құрылымдар мысалдарын қарастырайық. Мына 3.15,а, б-суреттерінде осындай графтардың төбелері көрсетілген. Бірінші жағдайда (а) есеп шешімі қарапайым кіші есептердің тізбегімен ауыстырыла алады. Екіншісінде (б) есеп шешімі кіші есептің бірімен ауыстырылады. Егер төбенің еншілес төбесі болмаса, онда ол – қарапайым есеп. Бастапқы есепке сай келетін *ЖӘНЕ/НЕМЕСЕ граф* төбесін *бастапқы төбе* деп атаймыз. Ал қарапайым есептер сипатына сәйкес келетін төбелерді *соңғы* (қорытынды) төбелер деп атаймыз. Енді *ЖӘНЕ/НЕМЕСЕ* графының төбелерінің ***шешілетін*** деп қандай жағдайда анықталатынын қарастырайық. Ол былайша анықталады:

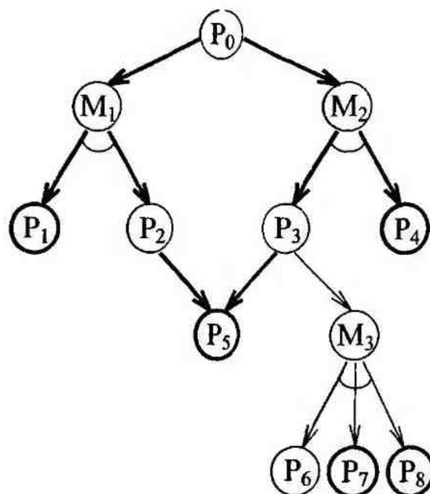
соңғы төбелер – ***шешілетін*** төбелер, өйткені олар қарапайым есептерге сай келеді;

- соңғы төбелер болмайтын НЕМЕСЕ-төбелер, ***шешілетін*** төбелер, ал оның тек еншілес төбелерінің бірі ***шешілетін*** төбе болғанда ғана бола алады;
- соңғы төбелер болмайтын ЖӘНЕ-төбелер ***шешілетін*** төбелер, ал оның әр еншілес төбесі ***шешілетін*** төбе болғанда ғана бола алады.

Егер іздеу кезінде бастапқы төбе ***шешілетін*** төбе болып табылса, онда қойылған есеп шешімі табылды деп есептеледі. Ол жолды Шешуші граф деп аталатын құрылым көрсетеді [39].

3.13-сурет. ЖӘНЕ-төбелер (a), НЕМЕСЕ-төбелер (b)

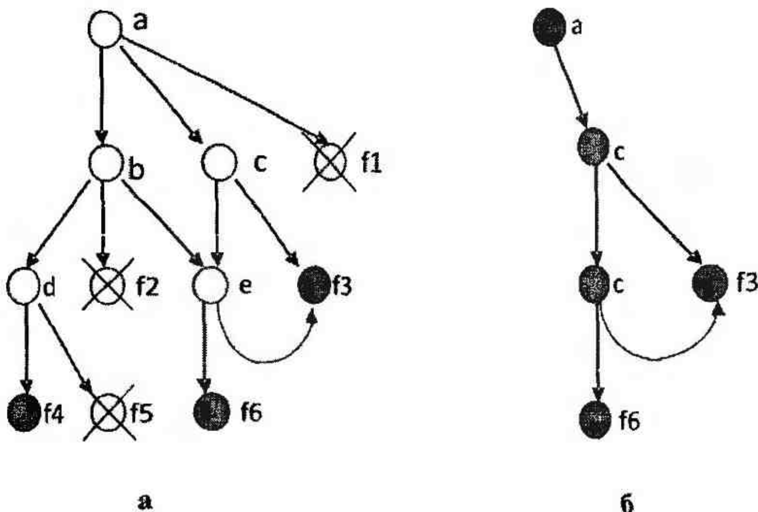
Шешуші граф – ЖӘНЕ/НЕМЕСЕ графының бір бөлігі, ол тек ***шешілетін*** төбелерден тұрады және бастапқы төбенің шешілетінін көрсетеді. Осындай *ЖӘНЕ/НЕМЕСЕ* графы және *Шешуші графтарға* мысалдар қарастырайық. Келесі 3.13,а-суретінде ЖӘНЕ-төбелер, ал (б) суретінде графтың НЕМЕСЕ-төбелері бейнеленген.



3.14-сурет. ЖӘНЕ/НЕМЕСЕ графына мысал

Ал мына графта (3.14-суретін қара) бастапқы төбе болып P_0 төбесі, ал соңғы төбелер есебінде – P_1, P_4, P_5, P_7, P_8 төбелері (олар қою сызықпен қоршалған) жүреді. Граф төбелеріндегі есеп шешімін табу үшін, редукция әдісін қолдануға болады. Бұндай *ЖӘНЕ/НЕМЕСЕ* графындағы іздеу максаты – бастапқы есептің ***шешілетінін*** көрсету, яғни бастапқы төбенің ***шешілетін*** төбе екенін көрсету. Оған жету үшін осы төбеге қатысы бар графтағы басқа төбелердің де шешілетін төбе екенін көрсете білу қажет.

Осы мысалдағы *ЖӘНЕ/НЕМЕСЕ* графындағы шешілетін төбелерге мыналар да жатады: M_1, M_2, P_2, P_3. Бұл графта екі шешуші граф бар: біріншісінің құрамында P_0, M_1, P_1, P_2 и P_5 төбелері бар; ал екіншісінде – P_0, M_2, P_3. төбелері бар. Логикалық программалау өкілі болып табылатын Пролог тілінің кез келген деректер қорындағы фактілер мен ережелері *ЖӘНЕ/НЕМЕСЕ* графының басқаша түрдегі жазбасы болып табылады. Мысал қарастырайық. Мысалы, мына 3.15,а-суреттегі «бұтаға» Пролог тіліндегі мына ереже сәйкес келеді: ***a : - b1, b2, ..., bn***. Ал келесі көрсетілген 3.15,б-суреттегі «бұтаға» Пролог тіліндегі бірнеше ереже дәл келеді: ***b : - c1. b : - c2 . ... b : - cm.*** Бұл суреттегі 3.15,а көрсетілген *ЖӘНЕ/НЕМЕСЕ* графының мына төбелері ***f3, f4, f6*** – шешілетін төбелер де, ал ***f1, f2, f5*** – шешілмейтін төбелер.



3.15-сурет. ЖӘНЕ/НЕМЕСЕ графы (а) және шешуші графы (б)

Аталған мысалдағы *ЖӘНЕ/НЕМЕСЕ* графына Пролог тіліндегі мына [14] Дерекқор сәйкес келеді:

a : - b.	**c : - e, f3.**	**f3.**
a : - c.	**d : - f4.**	**f4.**
a : - f1.	**d : - f5.**	**f6.**
b : - d, f2, e.	**e : - f6, f3.**	

Шешуші графты құру үшін, дерекқордағы әр предикатқа өз аргументтерін қоссақ, біз Пролог тілінде іске асырылатын қорды аламыз:

a (a(X)) : - b(X).	**c (c(X, Y)) : - e(X), f3(Y).**	**f3(f3).**
a (a(X)) : - c(X).	**d (d(X) : - f4(X).**	**f4(f4).**
a (a(X)) : - f1(X).	**d (d(X) : - f5(X).**	**f6(f6).**

b (b(X, Y, Z)) : - d(X), f2(Y), e(Z).
e (e(X, Y)) : - f6(X), f3 (Y).
Мақсатты тұжырым **? – a(X).**
Қайтаратын мәні **X = a(c(e(f6, f3), f3)).**

3.4.3. Предикаттарды құру арқылы іздеу әдістері

Жоғарыда келтірілген бірінші дәрежелі предикаттарды есептеу теориясындағы қорытындыларды тағы бір шолып өтелік. Бұл теориядағы өрнектердің негізгі класы есебінде ***ППФ (ДҚФ)*** өрнегі жүре алады. Егер осы ППФ = **true** болса, онда оны ***жалпымәнді*** деп атайды. Егер белгілі бір интерпретацияда ППФ true мәндері болса, онда осы интерпретация осы жиынды ***қанағаттандырады*** деп есептеледі. Егер ППФ «V» «~» таңбаларынан тұрса, онда оны ***сөйлем*** деп атайды. Белгілі бір ППФ-тан алынатын сөйлемді ***аксиома*** деп атайды. Егер әр сөйлемге оның терістелген сөйлемін қоссақ, онда ол үдерісті ***тавтология*** деп атаймыз. Егер белгілі бір тізімдер атаулары ***S*** сөйлемдер жиынына жатса, онда оны ***Эрбран универсумы*** деп атайды. Егер белгілі бір сала аумағына қатысты оның элементтерін белгілеуге Эрбран универсумы пайдаланылса және ол S жиынындағы барлық атомдық формулалар үшін қолданылса, онда оны S жиынына арналған ***Эрбран базасы*** деп атайды. Мысал ретінде жасанды зерде пәнінің классикалық есебі болып кеткен «Маймыл және банан» есебін қарастырайық. Бұл мәселенің шешуін есепті автоматты түрде анықтау жолдарын түсіндіру үшін қолданады. Есеп қойылымы: бөлмеде маймыл, жәшік және бір байлам банан бар. Байланған банан бумасы бөлме төбесіне ілінген және оның биіктігіне маймыл қанша секірсе

де, бойы жетпейді. Ол оған тек жәшік үстіне шықса ғана қолы жетеді. Маймылдың бананға қолы жететін іс-әрекеттер тізбегін құру қажет. Және есептің қойылымы бойынша маймыл бөлме ішінде ары-бері жүре алады, жәшікті қозғалта алады, жәшік үстіне шыға алады және оның үстінде тұрып бананға қол жеткізе алады.

Енді есепке қатысты қалыптарды кеңістіктегі осы сипатталған күйлерге байланысты қалай құру қажет? Сипатта мына күйлердің болуы міндетті: маймылдың бөлмедегі координаттары (тігінен, көлденеңінен), жәшіктің бөлмедегі координаттары (тігінен, көлденеңінен) және маймылда бананның болуы не болмауы. Бұл элементтерді (**w**, x, **y**, z) төрт элементтік тізім ретінде сипаттайды.

- **w** – маймылдың көлденең жазықтықтағы координаттары (екі өлшемді вектор),
- x – маймылдың жәшікте болу не болмауына байланысты 1 немесе 0 мәнін қабылдайтын айнымалы,
- **y** – жәшіктің тік жазықтықтағы координаттары (екі өлшемді вектор),
- z – маймылдың бананды алу не алмауына байланысты 1 немесе 0 мәнін қабылдайтын айнымалы.

Егер (**w**, x, **y**, z) тізімінің кез келген бір мәні бір күйді сипаттаса, онда бізде шексіз күйлер бар болар еді. Өйткені бөлмеде көптеген заттар мен олардың орналасуы және маймыл мен жәшіктің орындарын алмастыру нүктелері көп. Күйлер санын азайту үшін, бөлмедегі заттар саны аз деп қабылдап, оларды белгілі бір нүктелерге байлап ұстауға болады. Бірақ осылардың өзі күйлер санының кеңістігін үлкен жағдайда қалдырып отырады. Күйлер кеңістігін азайту мақсатында әртүрлі жолдар мен сызбанұсқалар қолданылады. Мұндай тәсілдердің біріне сызбанұсқаларды пайдалану жолын жатқызамыз. Бұл сызбанұсқада оған кіретін айнымалыларға белгілі бір мәндерді беру (мысалы, константа) арқылы іске асады. Онда бұл константалар операторды бір күйге қолданғанда пайда болатын күйлерге беріледі де, мақсаттың бар-жоғы тексеріледі.

Аталған есептегі операторлар есебінде маймыл орындайтын төрт іс-әрекетті пайдалануға болады. Оларға:

1. жақындау (**u**) – маймыл бөлме еденінің жазықтығындағы **u** нүктесіне қарай жылжиды (**u** – айнымалысы);
2. жылжыту (**v**) – маймыл бөлме еденіндегі жәшікті **v** нүктесіне қарай жылжытады (**v** – айнымалысы);
3. шығу – маймыл жәшікке шығады;
4. ұстау – маймыл бананды ұстап алады.

Құрамында айнымалылар болғандықтан, «жақындау», «жылжыту» операторлары іс жүзінде сызбанұсқалар болып табылады. Бұл операторлардың қолдану және әрекет жасау шарттары келесі көрсетілген ережелермен анықталады:

$$(w, 0, y, z) \xrightarrow{\text{жақындау (u)}} (u, 0, y, z),$$

$$(w, 0, w, z) \xrightarrow{\text{жақындау (v)}} (v, 0, v, z),$$

$$(w, 0, w, z) \xrightarrow{\text{шығу}} (w, 1, w, z),$$

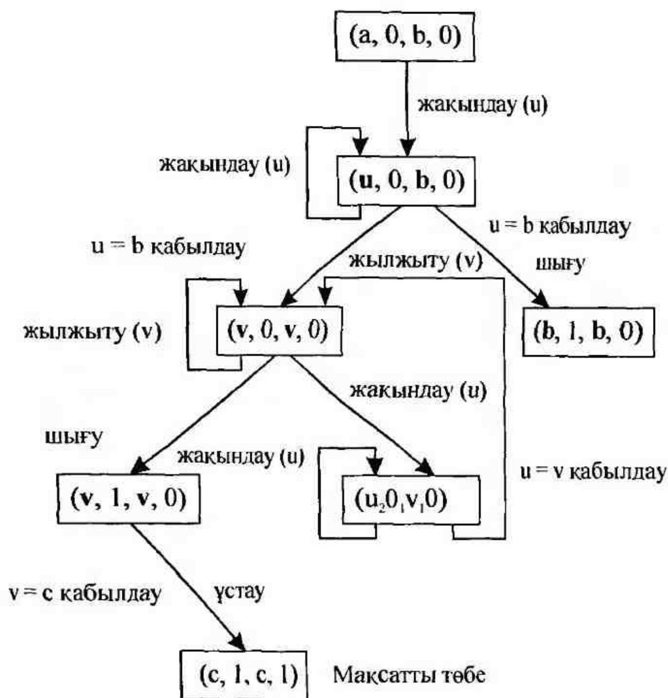
$$(c, 1, c, 0) \xrightarrow{\text{ұстау}} (c, 1, c, 1)$$

мұндағы **c** – бананның астындағы еден нүктелерін сипаттайтын координаттары (екі өлшемді вектор).

Кейбір операторларды қолдану, мысалы, «жылжыту» операторы айнымалы мәндеріне шектеуге тәуелді болуын көрсетеді.

Мысалы, маймыл координаттары мен жәшік координаттарының мәндері бірдей болуы кажет. Күйлерді сипаттайтын екі сызбанұсқаны ұқсас деп, егер олардағы тек айнымалылар атаулары басқаша болғандағы жағдайды есептейміз. Осындай қалыппен мақсатты күйлер жиынының элементтері соңғы элементі бір саны болып келетін кез келген тізіммен сипатталады.

Енді маймыл бастапқы сәтте еденнің **a** нүктесінде, жәшік **b** нүктесінде тұрды деп есептейік. Сонда бастапқы күй сипатын мынандай тізім түрі береді (**a**, 0, **b**, 0). Бұл күйдегі қолданылатын оператор – «жақындау», ол мына түрдегі (**u**, 0, **b**, 0) сызбанұсқаға әкеледі. Енді қолдануға болатын операторлар саны үшке жетті. Егер **u** = **b**, онда маймыл не жәшікке шығады, не оны жылжытады. Көрсетілген u айнымалысы қандай шамада болса да, маймылдың одан да басқа жерлерге жылжу мүмкіндігі бар. Маймылдың жәшікке шығуы мынандай күй (**b**, 1, **b**, 0) сипатына әкеледі және жәшіктің **v** күйіне жылжуы (**v**, 0, **v**, 0) сызбанұсқасына әкеледі және жаңа айнымалымен сипатталатын басқа орын күй сипатын өзгертпейді. Аталған барлық операторларды қолдану нәтижесінде белгілі бір түрдегі күйлер кеңістігін аламыз (3.16-суретті қара). Суретте көрсетілген граф өте үлкен емес, онда біздің есебімізге қатысты шешуші жол қою бағытталған сызықтармен белгіленген. Графтағы оператор айнымалыларының орнына олардың мәндерін қойсақ, мынандай операторлар тізбегін аламыз: ***жақындау (b), жылжыту (c), шығу, ұстау.***



3.16-сурет. Маймыл және банан есебінің графы

Енді бастапқы күйді сипаттайтын S_0 жиынының түрін келтірейік. Ол төрт түрлі ППФ-дан тұрады:

$$S_0=\begin{Bmatrix} \sim ONBOX \\ AT(G, b) \\ AT(O, a) \\ \sim HB \end{Bmatrix}.$$

Бұл жиындағы ONBOX предикаты T мәнін тек маймыл жәшік үстінде болғанда ғана қабылдайды; ал HB (HabenBanan) предикаты T мәнін тек маймыл бананды ұстап алғанда ғана қабылдайды; ал AT предикатының мағынасы айнымалылар белгілі бір мәндерде болғанда анықталады. Тағы бір айта кететін жайт: S_0 жиынындағы AT преди-

катында көрсетілген мына таңбалар: O – маймылды, ал G – жәшікті білдіреді. Максатты ДҚФ ретінде HB предикаты жүреді. Осы HB предикаты шығатын кез келген күйлер сипаты мақсатты күйге алып келеді. Сонымен, бізде күйлерге қолданылатын төрт оператор бар: «жақындау (u)», «жылжыту (v)», «шығу» және «ұстау». Алғашқы екі предикат – операторлық сызбанұсқалар; олардың нақты мәндері осы сызбанұсқалық айнымалылардың шамаларына байланысты болады. Әр операторды анықтағанда, олардың негізгі элементтері болып мыналар табылады: *қолдануға болатын Дұрыс құрылған формула* (ДҚФ), яғни ол осы операторды қолдануға болатын шарттарды сипаттайды, *ДҚФ жиындарын түрлендіру ережелері*, олар нәтижедегі күйлерді сипаттайды. Түрлендіру ережелерін де, қолдануға болатын ДҚФ-ты да, ДҚФ тізімдері ретінде беруге болады. Түрлендіру ережелерін тізімнен алып тастайды, ал қолдануға болатын ДҚФ ды тізімге қосып отырады. Сонымен бірге, егер кейбір ДҚФ-тар тізімнен алынбаса, онда олар бұрынғы күйлер жиынында қалып отырады. Енді осыдан соң, біздің аталған төрт операторымызды былайша анықтаймыз:

Жақындау (u)

қолдануға болатын ДҚФ: ~ONBOX

Түрлендірулер

Алып тастау:	AT (маймыл, $)
Қосу:	AT (маймыл, u).

Мұндағы «$» таңбасы кез келген терм орнында тұр. Ал Дұрыс құрылған формула AT (маймыл, $), яғни ДҚФ-ы «$» терм мәндеріне қарамастан алынып тасталуы қажет. Осы «жақындау (u)» предикаты операторлық сызбанұсқа болғандықтан, оны қолдану u айнымалысы бар күйлерді сипаттайтын сызбанұсқаны береді де, біз нақты күй мәнін аламыз.

Жылжыту (v)

қолдануға болатын ДҚФ: $\sim ONBOX \wedge (\exists x)[AT(O, x) \wedge AT(G, x)]$

Түрлендірулер

Алып тастау:	AT (маймыл, $)
	AT (жәшік, $)
Қосу:	AT (маймыл, v)
	AT (жәшік, v)

Шығу

қолдануға болатын ДҚФ: ~ONBOX∧(∃x)[AT(O, x)∧AT(G, x)]

Түрлендірулер

Алып тастау: ~ONBOX

Қосу: ONBOX

Ұстау

қолдануға болатын ДҚФ: ONBOX ∧ AT (ящик, c)

Түрлендірулер

Алып тастау: ~HB

Қосу: HB.

Операторлар арқылы жүзеге асатын ДҚФ ережелерін түрлендіруін анықтайтын алып тастау және қосу ережелерін қалыптастырғанда, үнемі есте сақталуға тиіс талаптар бар. Ол талаптарға мынаны жатқызамыз: алып тасталған ДҚФ ержелері ішінен алынбаған ДҚФ ережелерінен шығатыны болмауы қажет, өйткені онда осы алынып тасталған ДҚФ-тар басқа ережелерден тағы пайда болуы мүмкін. Енді мақсатты күйді іздеу үдерісі S_0 бастапқы күйлер жиынына операторларды қолдану арқылы өтетін стандартты іздеу үдерісі ретінде өтеді. Операторларды пайда болған күйлерге қолдану мақсатты HB предикаты алынғанша өтіп отырады да, ол мақсатты күй алынғанда тоқтайды. Сонымен, операторларды қолданатын операторлық сызбанұсқа тәсілімен өтетін алгоритм қадамдары бірнеше болады.

Іздеу үдерісінің бірінші қадамында HB предикаты S_0 жиынынан шығатын-шықпайтыны анықталады. Ол үшін дәлелдеуге қажетті ДҚФ ережесін терістейді де, қайшылықтарды шығаруға арналған резольвенцияға негізделген іздеу әдістерін пайдаланады. Ешқандай қарама-қайшылық $\{S_0\} \cup HB$ формуласынан шығарыла алмайтындықтан, біз (яғни ешқандай резольвенция мүмкін емес) мақсатқа жетудегі шарттарды тексеру теріс болды деген қорытынды жасаймыз.

Келесі қадамда қандай операторды қолданатынымыз туралы мәселені шешеміз. Бұл жерде тағы теоремаларды дәлелдейтін әдістердің бірі – резольвенция принципін қолдануға болады. Әр оператор үшін оның қолдануға болатын ДҚФ ережесі S_0 жиынынан шығатынын дәлелдесек жеткілікті болар еді. Осылайша біз «ұстау» операторының қолданылмайтынын көреміз. Қолданылуға болатын «шығу» және «жылжыту» операторларына арналған Дұрыс құрылған

формулалар бір-біріне ұқсас болып келеді. Бұл ДҚФ-тың сөйлемдер түріндегі терістеулері былайша өрнектеледі:

$$\sim\text{ONBOX} \vee \sim[AT(O, x) \vee AT(G, x)].$$

Бұл формуланың S_0 жиынымен қарама-қайшылығын табу да жетістік әкелмейді, сондықтан S_0 жиынына «шығу» және «жылжыту» операторларын қолдану тағы да сәтсіз аяқталады.

Енді S_0 жиынына «жақындау» операторын қолданамыз. Бұл қадамымыз сәтті болады да, біз түрлендіру ережелерін «жақындау (u)» операторына қолдану нәтижесінде S_1 жиынын аламыз. Одан ары қарай үдеріс жалғаса береді. Ең алдымен біз S_1 сызбанұсқасының жеке жағдайы бар-жоғын зерттейміз, яғни осы сызбанұсқадан мақсатты HB ДҚФ-сы шыға ма, соны анықтаймыз. Ол шықпайды. Одан ары қарай осы S_1 сызбанұсқасының жеке жағдайларына қолдануға болатын операторларды табамыз. Мысалы, «жылжыту (v)» операторын қолдануға болатынын тексеруде біз оның S_1 сызбанұсқасының жеке жағдайынан мына ДҚФ-сы

$$\sim\text{ONBOX} \wedge (\exists x)\ AT(\textit{маймыл}, x) \wedge AT(\textit{жәшік}, x)$$

шыға ма, соны анықтаймыз. Енді біз S_1 жиынындағы ***u*** айнымалысына ***b*** шамасын беру арқылы оған «жылжыту» операторын қолдануға болатынын көреміз. Бұл жеке жағдайды S_1' жиыны деп те белгілеуге болады. Алгоритмнің осы қадамында барлық ***u*** айнымалысы S_1 жиынындағы барлық ***u*** үшін ***b*** шамасына ауыстырылғанын қадағалау қажет. (Біздің мысалда енудің бір ғана шамасы бар, бірақ ол басқа есептер үшін көп болуы да мүмкін).

Енді S_1' жиынына «жылжыту» операторын қолдануға болатындықтан, осы жиынға «шығу» операторын да қолдануға болады. Дегенмен «ұстау» операторы S_1 жиынының бірде-бір жеке жағдайына қолдануға келмейді. Осылайша осы S_1 жиынына «жақындау» операторын да қолдануға болады, бірақ оны қолдану күйлер сипатын өзгертуге ешқандай әсер етпейді. Енді егер «жылжыту (v)» операторын S_1' жиынына қолдансақ, күйлер сипатының бір түрін, егер S_1' жиынына «шығу» операторын қолдансақ, күйлер сипатының екінші түрін алған болар едік. Бұл үдеріс мақсатты предикат қанағаттанғанша өтіп отырады. Соның нәтижесінде құрылған графтың бейнесін 3.16-суреттен көруге болады.

Енді осы жоғарыда сипатталып кеткен операторлар сызбанұсқалары қолдану тәсілінің алгоритм қадамдарын келтіре кетейік («Маймыл және банан» мысалы негізінде).

Іздеудің операторлық сызбанұсқалар тәсілінің алгоритм қадамдары.

1. HB предикаты S_0 жиынынан шыға ма, сол анықталады. Ол үшін алдымен S_0 жиынын құрамыз.

$$S_0 = \begin{Bmatrix} \sim \text{ONBOX} \\ \text{AT}(G,b) \\ \text{AT}(O,a) \\ \sim \text{HB} \end{Bmatrix}.$$

2. Қай операторды қолдануға болатынын шешеміз. Мұнда «ұстау» операторы жарамайды, ал «жылжыту» және «шығу» операторлары ұқсас және олардың терістеуі S_0 жиынан шықпайды. Сондықтан «жақындау» операторын қолданамыз. Соның нәтижесінде S_1 жиынын аламыз:

$$S_1 = \begin{Bmatrix} \sim \text{ONBOX} \\ \text{AT}(G,b) \\ \text{AT}(O,u) \\ \sim \text{HB} \end{Bmatrix}.$$

3. Енді осы S_1 сызбанұсқасынан мақсатты ДҚФ бар ма, соны анықтаймыз. Мақсат жоқ.
4. Осы S_1 жиынына «жылжыту» операторын қолданамыз. Сонда келесі S_2 сызбанұсқасын аламыз:

$$S_2 = \begin{Bmatrix} \sim \text{ONBOX} \\ \text{AT}(G,v) \\ \text{AT}(O,v) \\ \sim \text{HB} \end{Bmatrix}.$$

5. Енді «шығу» операторын қолданамыз. Сонда келесі S_3 күйін аламыз:

$$S_3 = \begin{Bmatrix} \text{ONBOX} \\ \text{AT}(G,v) \\ \text{AT}(O,v) \\ \sim \text{HB} \end{Bmatrix}.$$

6. Содан соң «ұстау» операторын қолданамыз. Соның нәтижесінде S_4 сызбанұскасын аламыз:

$$S_4 = \begin{Bmatrix} \text{ONBOX} \\ \text{AT}(G,v) \\ \text{AT}(O,v) \\ \text{HB} \end{Bmatrix}.$$

Бұл көрсетілген түрлендіру әдісін операторлық сызбанұсқалар тәсілі деп атайды. Ол бір ДҚФ жиынын екінші жиын түріне операторлар мен түрлендірулер көмегімен іске асырады.

3.5. Жасанды зерде есептерінің жаттығулары

Білімдерге негізделген жүйелердің басқару бөліктерін жобалағанда туатын маңызды мәселелер қатарына шешімді іздеу әдістерін, яғни шығару стратегияларын табу жолдарын жатқызуға болады. Таңдалып алынған іздеу әдістері ережелерді пайдалану жолдары мен оларды іске қосу тәртібіне әсер етеді. Таңдау процедурасы іздеу бағытын анықтау және оны іске асыру тәсілдерін табуға бағытталады. Шығару жолдарын басқару стратегиясын құрғанда, мынандай сұрақтарға жауап табу қажет:

1. Кеңістіктегі күйлердің қандай нүктесін бастапқы деп қабылдау керек? Өйткені білімдерге негізделген жүйе іздеуді бастамай тұрып, қай бағытта болса да, (тура және кері) іздеуді бір бастапқы нүктеден бастайды.

2. Шешімді іздеу тиімділігін қалай арттыруға болады? Шешімді іздеу тиімділігіне қол жеткізу үшін, талас тудыратын жолдарды шешетін эвристикаларды табу қажет. Талас тудыратын жолдар күйлер кеңістігінде бірнеше тармақталған жолдардың айырмаларында пайда болады. Олардың ішінде шешімге апармайтын тұйық жолдар бар.

Ойын программаларын құруға арналған жұмыстар «ойын теориясы» деген атаумен белгілі математикалық зерттеулерден сәл өзгеше түрде даму алды. Бұл екі бағыттың көздеген мақсаттары да әртүрлі еді. Ойындарды математикалық түрде талдау ең озық ойлы, өте аз қателесетін ойыншы стратегиясын зерттейді. Ал программалуда ойыншы іс-әрекетін ойын жолдарын басқару арқылы іздеу жолдарын қолдануға тырысады. Компьютерлік ойын программасын жазу үшін, ең алдымен ойын құрылымын бейнелеу проблемасын шешу қажет. Яғни құрылымды құратын ұғымдардың өзін анықтап алу қажет. Сондықтан ойын құрылымы шешімді іздеу әдістеріне тікелей әсер етеді. Ойындарға арналған есептердің екі түрлі аспектісі бар: бейнелеу және іздеу. Қазіргі кезде жасанды зерде зерттеулерінде ойын есептерін нақты түрде қалыптастыруға арналған әмбебап әдістер жоқтың қасы деуге болады. Дегенмен есеп мақсатын қоюға жарайтын белгілі бір жолдар бар. Ондай жолдар іздеу әдістерімен тікелей байланысты болады.

Іздеу әдістерінің әр түрлерін қарастырудан бұрын есеп мақсатын қоюға жарайтын бірнеше жолды атап өтсек:

1) кеңістіктегі күйлеріне негізделген жол;

2) үлкен есептерді кіші есептерге бөлшектеуге (Редукция) негізделген жол;

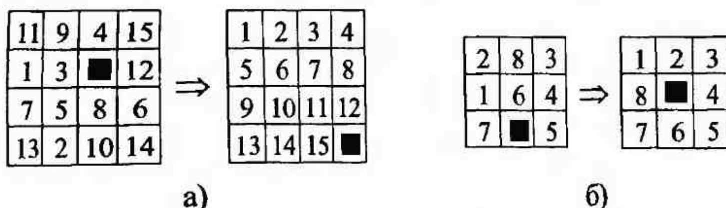
3) дәлелдеуге жататын теорема есебінде қарау жолы (1-ші дәрежелі предикаттарды есептеу).

Біз осы жолдарда қолданылатын бірнеше іздеу тәсіліне арналған жаттығуларды қарастырамыз: толық қарау әдісі; тереңнен іздеу әдісі; эвристикалық әдіс; редукция әдісі; предикаттарды есептеуге арналған әдіс.

3.5.1. «Соқыр әдістер» жаттығулары

Кеңістіктегі күйлеріне негізделген жол.

1. Бақылау мысалы. Есептің қойылымы. Мысал үшін «8» және «15» ойынын қарастырайық. Бастапқы күйлер 3x3 және 4x4 массивтері есебінде беріліп, мақсатты күйде де осы қалыптар тек басқа деректермен беріледі. Бұл күйлер түрлері 3.5.1,а,б-суреттерде көрсетілген.



3.5.1-сурет. «8» және «15» ойындарының сандар массивімен берілген бастапқы және мақсатты күйлері

Бұл мысалдағы *күйлерді* сипаттайтын қалып есебінде сандар массивтері берілген. Ал *оператор* есебінде бос клетканың солға, жоғары, оңға, төмен жылжуы берілген. Мысалдағы «8» ойынының тереңнен іздеу әдісі бойынша алынған ағаш түрі 3.5.2-суретте келтірілген. Мысалдағы төбелер олардың ашылу реті бойынша белгіленген. Және де шектелген терендік мәні 3-ке тең болып алынған (яғни мақсатқа жетудегі жолдардың ұзындығы үштен аз). Графтан біз тереңнен іздеу алгоритмінде іздеу алдымен бір жолдың бойынан өтетінін көреміз. Алдын ала көрсетілген тереңдік мәнінен соң іздеу одан қысқа болатын жолдар бойынан өтеді.

Осы әдістің іске асыру алгоритмін әртүрлі программалау тілдерінде қарастырайық. Іздеу алгоритмі былайша сипатталады: әр өтпеген төбе жанындағы өтілмеген көрші төбелер үшін мақсатты күй ізделеді. Мына *G = (V, E)* графы берілген дейік, мұндағы *V* – граф төбелерінің жиыны, *E* – граф қабырғаларының жиыны. Бастапқы уақыт кезеңінде графтың барлық төбелері *ақ* түсте дейік.

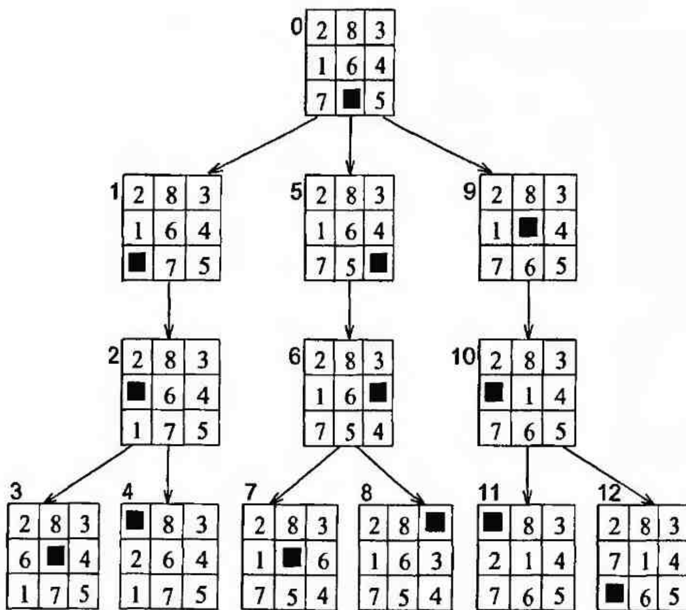
Мына іс-әрекеттерді орындаймыз:

1. Осы жиыннан *ақ түсті* төбелердің кез келгенін алып, оны v_1 деп белгілейік.
2. Ол үшін DFS(v_1) процедурасын орындаймыз.
3. Оны *қара түске* бояймыз.
4. Осы 1-3 дейінгі қадамдарды *ақ түсті* төбелер жиыны бос болғанша орындаймыз.

DFS процедурасы (параметрі – төбе u ∈ V)

1. Енді *u* төбесін *сұр түске* бояймыз.
2. Содан соң осы *u* төбесіне көршілес *w* төбесі үшін, мынандай екі қадамды орындаймыз:

- Егер *w* төбесі ақ түсте болса, DFS(w) процедурасын орындаймыз.
- Содан соң w төбесін *қара түске* бояймыз.



3.5.2-сурет. Тереңнен іздеудегі «8» ойынының графы

Іске асыру мысалдары. Java тілінде.

```
public static void search (Node node, Node goal) // Node – граф төбесі
{
    if (node.equals (goal))
    {
        System.out.println (node));
    }
    else
    {
        for (int i = 0; i < node.getNode().size(); i++) // Осы әдісті әр көрші төбеге шақырамыз
        {
            if (stack.add(node.getNode().get(i))) // Осы әдісті node.getNode().get(i) төбесіне шақырдық па, соны тексереміз
            {
                search(node.getNode().get(i), goal);
            }
        }
    }
}
```

C++ тілінде.

```
vector <vector <int>> graph;
vector <bool> used;
void dfs (int node_index)
{
    used[node_index] = true;
    for (vector<int>::iterator i = graph[node_index].begin(); i != graph[node_index].end(); ++i)
    {
        if ( !used[*i] )
            dfs(*i);
    }
}
```

Pascal тілінде.

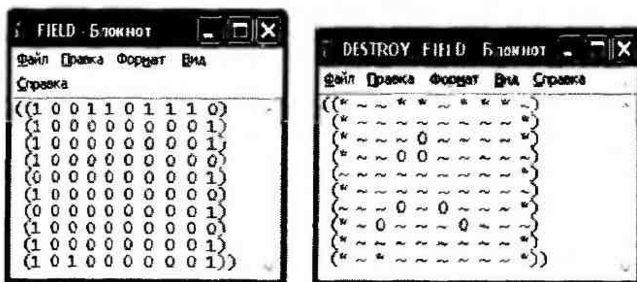
```
const
    MAX_N = 10;
var
    graph: array [1..MAX_N, 1..MAX_N] of boolean;
    visited: array [1..MAX_N] of boolean;
```

```
procedure dfs(v: integer);
var
    i: integer;
begin
    visited[v] := true;
    for i := 1 to MAX_N do
        if graph[v, i] and not visited[i] then
            dfs(i);
end;
```

2. Бақылау мысалы. Есептің қойылымы. Көпшілікке таныс «Теңіз шабуылы» ойыны бар. Компьютер «Теңіз шабуылы» ойынын сапалы түрде орындауына арналған алгоритмді құру қажет. Ойынның қосымша шарттары: әр жаңа ойын үшін компьютер жүрістері мен ойын күйлері әртүрлі болуы қажет. Ойында үш түрлі күйді атап өтуге болады:

1. Ойын алаңын кездейсоқ координаттар бойынша, кемелерге тигенге дейін ату, одан соң екінші күйге өту.
2. Алаңдағы тиген ұяшық төңірегін (тік немесе көлденең) ату, ол кеменің неше палубалы екенін анықтау үшін қажет, тиген соң, үшінші күйге өту.
3. Табылған бағытта кемені оны толығынан жойғанша ату, содан соң бірінші күйге өту.

Сонымен, ойын үш негізгі іс-әрекетке байланысты өрбиді: тигенге дейін ату, бағытты анықтау үшін ату, кеме толығынан жойылғанша ату.



3.5.3-сурет. «Теңіз шабуылы» ойынындағы тізіммен берілген күйлер

Осы ойынға арналған күйлер қалыптарын қарайық. Қалыптар тізімдер түрінде беріледі. Оның түрі 3.5.3,а, б-суреттерінде келтірілген.

а) бастапқы күй б) мақсатты күй

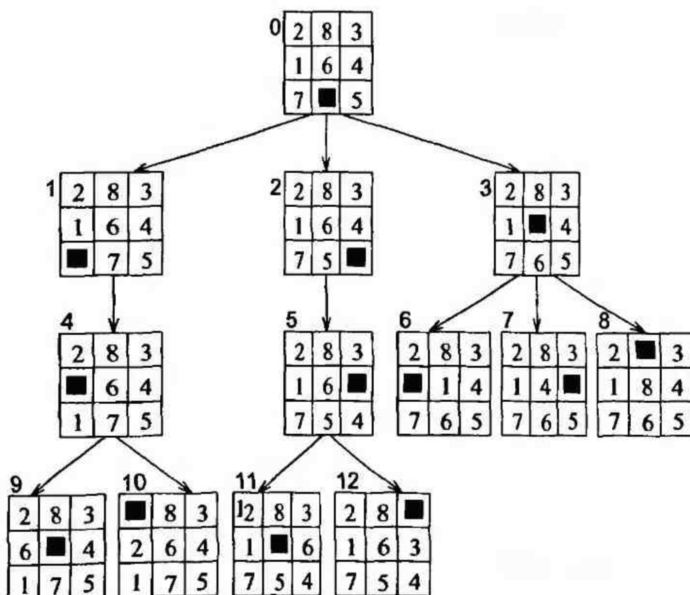
Осы күйлер қалыптарының тізімін функционалдық программалаудың тілі Лисп кодасында былайша бейнелейміз:

(setq input_stream (open «d:field.txt»:direction:input)); мәтіндік файлды оқу үшін ашамыз.

(defun set_missing_comp(lst i j ip jp)); кемелер түрлерін тізім түрінде оқитын функцияны анықтаймыз.

Бұл ойынның математикалық үлгілерінің бірі кемелерді анықтауға ықтималдылықты есептеу арқылы жететін жолды пайдаланады. Кемелер орналасуы кездейсоқ болады және оларды іздеу жолдары да әртүрлі болып келеді.

3. Бақылау мысалы. Есептің қойылымы. Мысал үшін тағы да сол «8» ойынын қарастырайық. Осы ойынға жайылып іздеу әдісін қолданғандағы пайда болатын ағаш түрі 3.5.4-суретте келтірілген.



3.5.4-сурет. Жайылып іздеудегі «8» ойынының графы

Әдістің іске асырылуы. Жайылып іздеуде «кезек» деп аталатын құрылым пайдаланылады. Ол үшін кезекке бастапқы төбені енгіземіз. Содан соң кезек бос болғанға дейін жұмыс істейміз. Ол былайша өтеді: кезектегі элементті алып, оған оның әлі қарастырылмаған көршілес барлық элементтерін қосамыз да, содан мақсатты күйді іздейміз.

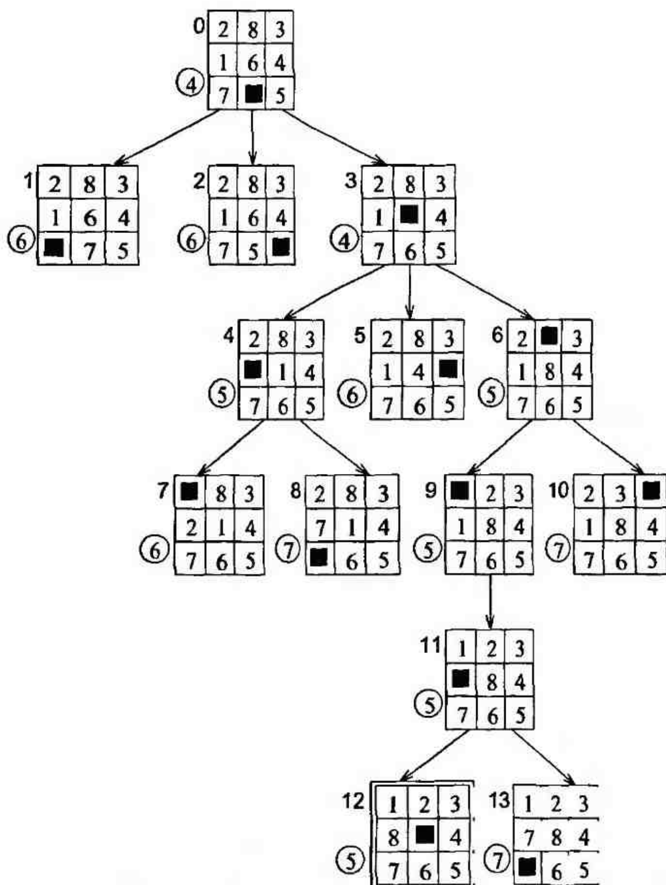
Кез келген графқа арналған Python тіліндегі кодалар. Жайылып іздеудің функциясы BFS(G, s) аргументі ретінде G графын аламыз. Ол [N, E] тізімі түрінде беріледі, мұндағы N – графтағы түйіндер саны, E – қабырғалар тізімі, ол [[a, b], [b, c]...] түрінде болады және ондағы түйіндер қатармен де, санмен де берілуі мүмкін, және s – іздеуді содан бастайтын төбе аты. Аталған функция екі тізімнен тұратын кортежді қайтарады: p – жайылып іздеу ағашының түйіндеріне арналған аталық элементтердің атауын білдірсе, d – s түйініне дейінгі қашықтықты білдіреді.

Кез келген графқа арналған Python тіліндегі кодалар.

```
def BFS(G, s):
    color, d, p = {}, {}, {}
    for u in [x for x in range(G[0]) if x != s]:
        color[u], d[u], p[u] = «white», «inf», None
    color[s], d[s], p[s] = «gray», 0, None
    Q = [] # queue
    Q.append(s)
    while Q != []:
        u = Q.pop(0)
        for v in [x[1] for x in G[1] if x[0] == u]:
            if color[v] == «white»:
                color[v], d[v], p[v] = «gray», d[u] + 1, u
                Q.append(v)
        color[u] = «black»
```

Эвристикалық әдіс (Тең бағалар әдісі). Бұл тәсіл бойынша бастапқы төбеден мақсатты төбеге дейін бағасы ең арзан болатын төбе арқылы өтетін жол таңдалып алынады. Сонымен бірге жолдың өзінің ұзындығын да бағалап отыру тәсілі бар. Тең бағалар әдісінде $c(n_i, n_j)$ деген бағалау функциясы беріледі, ол $\boldsymbol{n_i}$ төбесінен одан кейінгі $\boldsymbol{n_j}$ төбесіне өтуде төбелердің бағасын есептейді. Бұл әдісте құрылатын ағаш төбелерінің келесі төбеге өтудегі бағасы шығарылып еске сақталады да, ***s*** төбесінен ***n*** төбесіне дейінгі жол белгіленіп отырады. Мысалы ***g(n)*** құрылған ағаштағы ***s*** төбесінен ***n*** төбесіне дейінгі жол бағасы болсын. Осы g(n) функциясының ең арзан болатын жолын таңдау арқылы мақсатты күйге жететін ең қысқа жолды таңдап аламыз.

4. Бақылау мысалы. Есептің қойылымы. Мысал үшін тағы да сол «8» ойынын қарастырайық. Осы ойынға эвристикалық іздеу әдісін қолданғандағы пайда болатын ағаш түрі 3.5.5-суретінде келтірілген.



3.5.5-сурет. Эвристикалық іздеудегі «8» ойынының графы

Сонымен бірге тең бағалар әдісі бойынша жұмыс істейтін алгоритмді ұзындығы ең қысқа болатын жолды табу үшін де пайдалануға болады. Ол үшін әр қабырғаның ұзындығының бағасын бірге тең деп алуға болады. Қорытындылай келсек, бұл тәсілде мынандай бірлік функциясы таңдап алынады: $f(n) = g(n) + w(n)$ – ол әр төбеге берілетін бағаны есептейді, мұндағы $g(n)$ – жол ұзындығы, $g(0) = 0$ деп қабылданып алынады, ал $g(n) = g(n-1) + 1$, сонда $g(1) = 0 + 1 = 1$ болады. Функциядағы $w(n)$ – мақсатты күймен салыстырғанда өз орнында жатпаған фишкалар саны. Іздеу кезінде ең арзан төбені таңдап, одан ары қарай іздеуді сол бағытта жүргіземіз. Мысалдағы бағалар 3.5.5-суретінде көрсетілген.

Ол бағалар төбелер үшін мынандай болып келеді: *f(0) = 4, f(1) = 6, f(2) = 6, f(3) = 4, f(4) = 5, ..., f(12) = 5, f(13) = 7.* Ең соңында мақсатты күйге апаратын жолды көрсететін ең арзан төбе таңдалып алынады.

Жоғарыда аталған ойынның күйлер графы біраз мөлшерде болады. Сондықтан оны шешуде эвристикалық тәсіл өте қажетті құралдардың біріне жатады. Енді осы алгоритмнің «8» ойынына қатысты Лисп тіліндегі кодаларын келтірейік [39]. Лисп тілінде құрылған негізгі функция атауы HEURISTIC_SEARCH деп аталады. Осы Лисп функциясында қолданылатын қосымша функцияларға: IS_GOAL, EST, OPENING, SOLUTION, CONNECT жатады. Ойын күйлері тізім болады. Ол келесі элементтерден тұрады:

- күйлер идентификаторы, онда кіріктірілген gensym Лисп функциясының көмегімен пайда болатын S1, S2, S3, ... атомдары пайдаланылады;
- күйлерді сипаттау – квадрат қатарларына біртіндеп жазылған фишкалар нөмірлерінің тізімі;
- іздеу ағашындағы күйлер-төбелердің тереңдік саны;
- күйлерді бағалайтын эвристикалық сандық баға.

Күйлер сипатына тізімнің бірінші элементі есебінде белгілі бір күйді тудыратын бос клетканы жылжыту операторын қабылдайды. Бұл элемент төбелерді ашу кезінде туатын ұқсас күйлердің қайталануын болдырмау үшін қажет. Бұл операторлар мынандай: right, left, up, down деген атом-атаулармен, ал бос клетка – мына # таңбамен белгіленеді.

Мысалы, мына тізім (S3 ('left 2 8 3 1 6 4 # 7 5) 1 6) S3 күйін сипаттайды. Ол бос клетканы солға жылжытудан алынған және оның эвристикалық бағасы 6-ға тең, осыған сәйкес іздеу ағашындағы төбе 1 тең тереңдікте орналасады. Тағы мына жағдайды атап өтуіміз керек. Эвристикалық баға шамасы тек эвристикалық алгоритмдерде қолданылса, ал төбе тереңдігінің шамасы эвристикалық алгоритмдермен қоса, шектелген тереңнен іздеу алгоритмдерінде пайдаланылады.

Төменде келтірілген «8» ойынына қатысты Лисп функцияларын «15» ойынына да қолдануға болады. Ойын квадратының жақтарының мөлшері Size орасан үлкен айнымалысында сипатталады. Ал мына Goalstate орасан үлкен айнымалысында мақсатты күй (идентификатор, тереңдік және эвристикалық баға жоқ) сақталады. Бағалау EST функциясының мәні «өз орнында» тұрмаған фишкалар санының және іздеу ағашының бағаланатын күй-төбесіне дейінгі тереңдік жолының ұзындығының қосындысы ретінде анықталады.

Енді осы алгоритмнің Лисп тіліндегі кодаларын келтірейік.

```
; «8» ойынының эвристикалық алгоритміндегі Лисп функциясы;
   (defun IS_GOAL (State)
      (equal (cdadr State) Goalstate ))
   (defun EST (S)
      (prog (Len G N)
      (setq Len (cadr S)) (setq N 0)
      (setq S (cdar S)) (setq G Goalstate)
; берілген және мақсатты күйлердің
бір мезетте тізім-сипаттарын қарау:;
   ES (cond ((null S) (return (+ N Len)) ))
      (cond ((neq(car S )(car G )) (setq N (add1 N))))
        (pop S) (pop G) (go ES) ))
(defun OPENING (State)                ; қосымша функцияны анықтау;
(prog (Op St Dlist K I J El)
; күйлер сипаттарының құрама элементтерін бөлектеу :;
      (setq St (cadr State))
      (setq Op (car St)) (setq St (cdr St))
      (setq State St) (setq K 0)
; тізімдегі «бос клетка»ның К нөмірін іздеу:;
   OP (setq El (car St))
      (setq K (add1 K))
      (cond ((neq El '#) (setq St(cdr St)) (go OP)))
; «бос клетка»ның қатар және баған нөмірін есептеу:;
      (setq J (/ K Size))
   (setq I (add1 (- J (rem J)))) (setq J (rem K Size))
; «бос клетка»ның кезекепен жылжуын тексеру;
; оңға/солға/жоғары/төмен (Op операторын таладу арқылы);
; іздеу ағашында ұқсас циклдарды алып тастаймыз (қайтару);
; екі операторды қолданған соң бастапқы күйге қайтып келу:;
      (cond ((and (neq Op 'left) (< J Size))
 (ADD_STATE 'right K (add1 K)) ))
      (cond ((and (neq Op 'right) (> J 1))
 (ADD_STATE 'left K (subl K)) ))
      (cond ((and (neq Op 'down) (> I 1))
 (ADD_STATE 'up K (- K Size)) ))
      (cond ((and (neq Op 'up) (< I Size))
 (ADD_STATE 'down K (+ K Size)) ))
      (return Dlist) ))
```

Төбелерді ашқанда мынандай ADD_STATE қосымша функция пайдаланылды. Ол балалық төбелер тізіміне жаңа төбенің тізім сипатын қосады. Бұл тізім екі элементтен тұрады: осы төбенің идентификаторы (оны gensym функциясы тудырады) және фишкалар нөмірлерінің тізімі. Осы ADD_STATE қосымша функциясында кіріктірілген NTH лисп функциясы пайдаланылады. Оның атқаратын қызметі: берілген тізімнен анықталған реттік нөмірі бар элементті таңдап алады. Элементтер есебі нөлден басталады.

```
; қосымша функцияны анықтау;
(defun ADD_STATE (Op K1 K2)
   (push(list (gensym)
     (cons Op
       (cond((< K1 K2)
            (EXCHANGE State '# (NTH (sub1 K2) State)))
       (t (EXCHANGE State (NTH (sub1 K2) State)'#))) ))
   Dlist))
```

Осы қосымша ADD_STATE функциясында EXCHANGE рекурсивті функциясы пайдаланылады. Ол ойынның жаңа күйін List бастапқы күй тізімнің берілген элементтерін алмастыру арқылы қалыптастырады.

```
; рекурсивті функцияны анықтау;
   (defun EXCHANGE (List Elem1 Elem2)
(cond((eql Elem1 (car List))
   (cons Elem2 (EXCHANGE (cdr List) Elem1 Elem2)))
((eql Elem2 (car List)) (cons Elem1 (cdr List)))
   (t (cons(car List)(EXCHANGE (cdr List) Elem1 Elem2)))))
```

Тағы бір қосымша CONNECT функциясы Curr ағымдағы күй төбелерінен берілген балалық күй төбелеріне (олар Dlist тізімінде) бағытталған нұсқағыштар тізімін қалыптастырады. Әр нұсқағыш аталық төбе идентификаторы, балалық төбе идентификаторы және оларды байланыстыратын оператор атуынан тұратын үш элементті тізім.

```
; қосымша функцияны анықтау;
(defun CONNECT (Dlist Curr)
   (prog (D Di Ci Rlist)
   C (setq D (car Dlist)) (setq Dlist (cdr Dlist))
     (setq Di (car D)) (setq Ci (car Curr))
     (setq Rlist (cons (list Ci Di (caadr D)) Rlist))
     (cond ((null Dlist) (return Rlist)))
     (go C)))
```

Тағы бір қосымша SOLUTION функциясы шешуші жолды ерекшелей отырып, бастапқы күйді максатты күйге түрлендіретін операторлар (олардың атауларын) тізбегін құрады. Мақсатты күй аргументі Reflist аргументі – күйлер арасындағы барлық нұсқағыш байланыстардың тізімі. Шешуші жолдағы кезекті төбеге арналған нұсқағышты іздеу үшін, қосымша LOOK_FOR функциясы пайдаланылады.

```
; қосымша функцияны анықтау;
(defun SOLUTION (Goal Reflist)
   (prog (Sollist        ;шешім құрылатын тізім;
      Gi Edge)
        (setq Gi (car Goal))
      S (cond((eq Gi 'S0) (return Sollist)))
        (setq Edge (LOOK_FOR Gi Reflist))
        (setq Sollist (cons (caddr Edge) Sollist))
        (setq Gi (car Edge))
        (go S)))
; қосымша функцияны анықтау;
(defun LOOK_FOR (Id List)
    (cond ((null List) (quote error))
      ((eq ID (cadar List)) (car List))
      (t (LOOK_FOR Id (cdr List)))))
```

Енді эвристикалық іздеу жолымен «8» ойынын іске қосатын соңғы функцияны келтіреміз. Оның ең алдында кіріктірілген gensym функциясының айнымалылары-параметрлері тағайындалады.

```
; іске қосу функциясын анықтау;
(defun main ()
    (prog(Goalstate Initstate Size)
      (setq *gensym-prefix* 'S)
      (setq *gensym-count* 1)
      (setq Size 3)
      (setq Goalstate '(1 2 3 8 # 4 7 6 5))
      (setq Initstate '(? 2 8 3 1 6 4 7 # 5))
      (HEURISTIC_SEARCH Initstate) ))
```

Осы жаттығуларды орындау нәтижесінде мына терминдерді түсініп, оларды нақты есептерге қолдана білу қажет: ***күйлер, күйлерді сипаттайтын қалыптар, оператор, бастапқы күй сипаты, мақсатты күй сипаты, граф түріндегі жазба, төбелерді ашу, тереңнен іздеу әдісі*** (Depth-first search, ***DFS***), ***іздеу әдістеріндегі open және closed***

тізімдері, жайылып іздеу әдісі (***BFS***, Breadth-first search), ***кезек құрылымы*** (***FIFO*** «first-in-first-out»), ***стек құрылымы*** (***LIFO*** «last-in-first-out»), ***эвристикалық іздеу әдісі.***

Тапсырмалар:

1. Тереңнен іздеу әдісіне негізделген компьютерлік ойын бағдарламасын құру.
2. Жайылып іздеу әдісіне негізделген компьютерлік ойын бағдарламасын құру.
3. Эвристикалық іздеу әдісіне негізделген компьютерлік ойын бағдарламасын құру.
4. Іздеуді деректерден бастайтын әдіске негізделген компьютерлік ойын бағдарламасын құру.
5. Іздеуді мақсаттан бастайтын әдіске негізделген компьютерлік ойын бағдарламасын құру.
6. Іздеуді «сараң» деген атпен белгілі алгоритм әдісіне негізделген компьютерлік ойын бағдарламасын құру.
7. Іздеуді «Минимакс» деген атпен белгілі алгоритм әдісіне негізделген компьютерлік ойын бағдарламасын құру.

3.5.2. Редукция әдісінің жаттығулары

Редукция әдісіне арналған жаттығу үшін тағы да сол классикалық болған Ханой мұнарасы есебін келтіруге болады (3.4-параграфын қара).

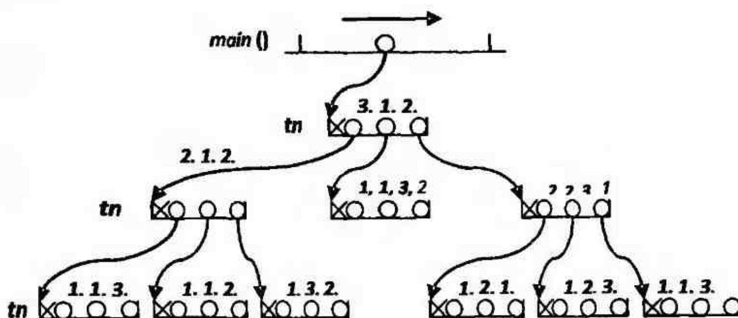
Бақылау мысалы. Есептің қойылымы. **Мысал үшін жоғарыда аталған Ханой мұнарасы есебін қарастырамыз.**

Есептің ең қарапайым жағдайы үшін, яғни пирамидада тек бір диск болғанда, бір ғана амал орындалады – дискіні ***i*** қазығынан ***j*** қазығына қою, (бұл ауыстыруды былайша белгілейміз ***i*** → ***j***). Жалпы жағдайда саны ***n*** болатын дискілерді ***w*** қазығын қосымша ретінде пайдалана отырып, ***i*** қазығынан ***j*** қазығына қою қажет (1.2.8-суретін қара). Ең алдымен ***n-1*** дискілерді ***i*** қазығынан ***w*** қазығына қою керек, мұнда ***j*** қазығын қосымша ретінде пайдаланып отырады. Содан соң бір дискіні ***i*** қазығынан ***j*** қазығына қою қажет, соңынан ***n-1*** дискілерін ***w*** қазығынан ***j*** қазығына ***i*** қазығын қосымша ретінде пайдаланып ауыстырамыз. Сонымен, ***n*** дискілерді ауыстыру есебі екі ***n-1*** дискілерін ауыстыру есебі мен бір қарапайым есепке бөлінеді. Оны тізім ретінде былайша өрнектейміз:

T (n, i, j, w) = T (n 1, i, w, j), T (1, i, j, w), T (n 1, w, j, i).

Ханой мұнарасы жайлы есепте ***tn (n, i, j, w)*** ішкі рекурсивті процедурасы пайдаланылады. Мұндағы саны ***n*** болатын дискілерді ***w*** қазығын қосымша ретінде пайдалана отырып, ***i*** қазығынан ***j*** қазығына қою қажеттігі тізім түрінде берілген. Яғни тізім түрі {i, j, w} = {1,3,2}.

Осы ***tn*** процедурасының m = 3 болғандағы бірінен соң бірін шақыру жолдары суретте бейнеленген (3.5.6-суретін қара). Әрдайым ***tn*** процедурасын шақырғанда, ***n, i, j, w*** параметрлеріне арналып компьютер жадынан орын бөлінеді және процедурадан қайтатын орын реті еске сақталып тұрады. Содан соң ***tn*** процедурасынан қайта қайтқанда ***n, i, j, w*** параметрлеріне одан бұрынғы шақыру кезіндегі бөлінген жады орны босатылады да, басқару қайтару нүктесіне беріледі. Суреттегі дөңгелекпен белгіленген – ***tn*** функциясын шақыру, ал ***x*** – рекурсиядан шығу шартын тексеру және ауыстыруды экранға шығару. Рекурсивті функцияларды басқаруға «стек» механизмін пайдаланады. Оны әдебиетте ***LIFO*** «last-in-first-out», яғни «соңғы келді – бірінші қызмет көрсетілді» құрылымы деп атайды. Бұл механизмде күйлер тізімнің сол жағынан қосылып, сол жағынан өшіріледі.



3.5.6-сурет. Ханой мұнарасына арналған стек

Осы алгоритмнің бірнеше тілде іске асқан кодаларын қарастырайық.

C++ тілінде

```
#include <stdio.h>
void tn (int, int, int, int); /* функция */
main() /* вызывающая */
{
int n;
scanf («%d»,&n);
tn (n, 1, 2, 3);
```

```
}
void tn (int n, int i, int j, int w) /* рекурсивная */
{
if (n>1) /* функция */
{
tn (n 1, i, w, j);
tn (1, i, j, w);
tn (n 1, w, j, i);
}
else
printf (» \n % d ->%d», i, j);
return;
}
```

Осы алгоритмнің іске асуын Пролог тілінің кодаларымен өрнектейік.

```
Domains
Loc = right;middle;left.        */аргумент типтеріне мағыналы
                                  атаулар береміз

Predicates                      */осы предикат аргументтері қандай
                                домендерге (типтерге) жататынын
                                хабарлаймыз
Hanoi (integer) – procedure(i).
move (integer, loc, loc, loc) – procedure(i, i, i, i).
inform (loc, loc) – procedure(i, i).

clauses                         */ойын мақсаты
Hanoi (N) : –
   move (left, middle, right).
move (1, A, _ , C) : –
   inform (A,C), !.

move (N, A, B , C) : –
N1 = N – 1,
move (N1, A, C , B),            */бір қазықтан келесіге N дискілерді
                                ауыстырғанда, үшінші қазықты
                                қосымша қолдана отырып, предикат
                                (move) предикатын пайдаланады
inform (A, C),
move (N1, B, A, C).
```

inform (loc1, loc2) : – **/нақты дискімен орындалатын амалды (inform) предикаты атқарады*

write («move a disk from», «loc1», to «loc2»), nl.

Goal **/мақсат бөлігінде (Goal) ойнайтын дискілер саны көрсетіледі*

Hanoi(3).

Осы алгоритмнің Лисп тіліндегі кодасын құрудан бұрын тізімдердің жұмыс істеуін бақылайтын мысалға көңіл аударайық.

1	*> (hanoi-tower 4)*
2	*Қазықтардың ауыстыруға дейінгі күйі:*
3	*a: (1 2 3 4)*
4	*b: ()*
5	*c: ()*
6	*'a' қазығынан 'b' қазығына:*
7	*a: (2 3 4)*
8	*b: (1)*
9	*c: ()*
10	*'a' қазығынан 'c' қазығына:*
11	*a: (3 4)*
12	*c: (2)*
13	*b: (1)*
14	*'b' қазығынан 'c'қазығына:*
15	*b: ()*
16	*c: (1 2)*
17	*a: (3 4)*
18	*'a' қазығынан 'b' қазығына:*
19	*a: (4)*
20	*b: (3)*
21	*c: (1 2)*
22	*'c' қазығынан 'a' қазығына:*
23	*c: (2)*
24	*a: (1 4)*
25	*b: (3)*
26	*'c' қазығынан 'b' қазығына:*
27	*c: ()*
28	*b: (2 3)*

```
29 | a: (1 4)
30 | 'a' қазығынан 'b' қазығына:
31 | a: (4)
32 | b: (1 2 3)
33 | c: ()
34 | 'a' қазығынан 'c' қазығына :
35 | a: ()
36 | c: (4)
37 | b: (1 2 3)
38 | 'b' қазығынан 'c' қазығына:
39 | b: (2 3)
40 | c: (1 4)
41 | a: ()
42 | 'b' қазығынан 'a' қазығына:
43 | b: (3)
44 | a: (2)
45 | c: (1 4)
46 | 'c' қазығынан 'a' қазығына:
47 | c: (4)
48 | a: (1 2)
49 | b: (3)
50 | 'b' қазығынан 'c' қазығына:
51 | b: ()
52 | c: (3 4)
53 | a: (1 2)
54 | 'a' қазығынан 'b' қазығына:
55 | a: (2)
56 | b: (1)
57 | c: (3 4)
58 | 'a' қазығынан 'c' қазығына:
59 | a: ()
60 | c: (2 3 4)
61 | b: (1)
62 | 'b' қазығынан 'c' қазығына:
63 | b: ()
64 | c: (1 2 3 4)
65 | a: ()
66 | соңы
```

Осы жаттығуларды орындау нәтижесінде мына терминдерді түсініп, оларды нақты есептерге қолдана білу қажет: ***қарапайым есептер, есеп редукциясы, редукция операторы, бастапқы күй сипаты, мақсатты күй сипаты, граф түріндегі жазба, ЖӘНЕ/НЕМЕСЕ графы. Шешуші граф, ЖӘНЕ-төбе, НЕМЕСЕ-төбе, соңғы төбелер, шешілетін төбелер, кезек құрылымы*** (***FIFO*** «first-in-first-out»), ***стек құрылымы*** (***LIFO*** «last-in-first-out»).

Тапсырмалар:

1. Күйлері ЖӘНЕ/НЕМЕСЕ графымен сипатталатын ойын программасын құру.
2. Редукция әдісін пайдаланатын компьютерлік ойын программасын құру.
3. Эвристикалық алгоритмді пайдаланатын компьютерлік ойын программасын құру.
4. Тура іздеу стратегиясын пайдаланатын компьютерлік ойын программасын құру.
5. Мақсаттан іздеу стратегиясын пайдаланатын компьютерлік ойын программасын құру.

3.5.3. Предикатты есептеу тәсілінің жаттығулары

Бұл жаттығуда 1-дәрежедегі предикаттарды есептеу теориясындағы әдістерді пайдалана отырып, күйлерді ЖӘНЕ\НЕМЕСЕ графын пайдалану арқылы ойындарға арналған компьютерлік зерделік программа құру жолдарын қарастырамыз.

Енді көпшілікке кеңінен таныс маймыл мен банан жайлы классикалық есеп түрін қарастырайық. Оның сөйлеммен берілетін қойылымы былайша өрнектеледі: бөлмеде маймыл, жәшік және бір байлам банан бар. Байланған банан бумасы бөлме төбесіне ілінген және оның биіктігіне маймыл қанша секірсе де бойы жетпейді. Ол оған тек жәшік үстіне шықса ғана қолы жетеді. Маймылдың бананға қолы жететін іс-әрекеттер тізбегін құру қажет. Және есептің қойылымы бойынша маймыл бөлме ішінде ары бері жүре алады, жәшікті қозғалта алады, жәшік үстіне шыға алады, және оның үстінде тұрып бананға қол жеткізе алады. Бұл есептің теориялық қойылымын [24] әдебиеттегідей біз 3.4.3-параграфында қарастырып өткенбіз. Біз қазір оның Пролог тіліне ыңғайластырылып қойылған жобасын зерттейміз [39].

Есеп күйін сипаттау үшін мына мағлұматтар қажет: бөлмедегі маймылдың орны – еден жазықтығының көлденеңінде ме, тігінде ме (яғни маймыл еденде тұр ма, әлде жәшікте ме), бөлімдегі жәшіктің едендегі орны және маймылдын қолында банан бар ма, әлде жоқ па. Осының бәрін төрт элементтік тізім түрінде беруге болады: (*ПолОб, ВертОб, ПолЯщ, Мақсат*),

Мұндағы:

ПолОб – маймылдың едендегі күйі (ол жазықтықтың екі элементтік векторлық координаты бола алады);

ПолЯщ – маймыл мен жәшіктің едендегі күйі;

ВертОб – маймылдың еденде немесе жәшікте болуына байланысты П немесе Я таңбасы қойылады;

Мақсат – маймылдың бананға қолы жетті ме, жоқ па, соған байланысты 0 немесе 1 саны.

Сонымен бірге еден жазықтығындағы мәні бар мына үш нүктені де қарастырамыз:

T_O – маймылдың ең алғашқы орнының нүктесі;

$T_Я$ – жәшіктің ең алғашқы орнының нүктесі;

$T_Б$ – банан бумасы орналасқан нүкте.

Сонда есептің бастапқы күйі мына тізіммен (T_O, П, $T_Я$, 0) сипатталады да, ал мақсатты күй (T_O, П, $T_Я$, 1), яғни соңғы элементі 1-ге тең болатын тізіммен сипатталады.

Есептегі операторлар былайша анықталады:

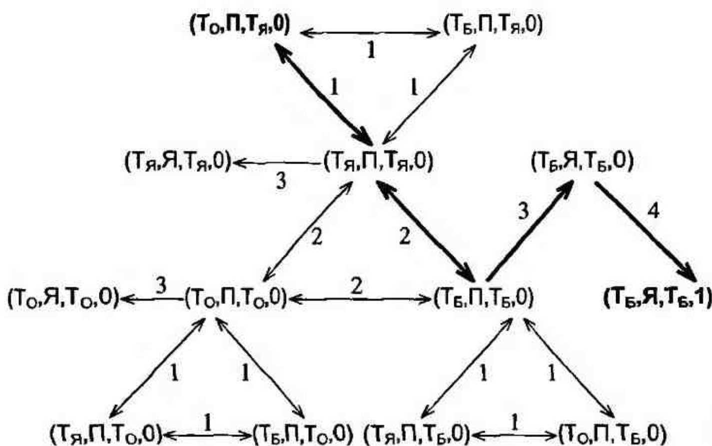
1) *Өту (W)* – маймыл еден жазықтығындағы *W* нүктесіне ауысады;
2) *Қозғау (V)* – маймыл жәшікті еденнің *V* нүктесіне жылжытады;
3) *Шығу* – маймыл жәшікке шығады;
4) *Ұстап алу* – маймыл банан бумасын ұстап алады.

Осы аталған төрт оператордың қолдану шарты мен іс-әрекетін келесі продукциялар ережелері бойынша жазуға болады:

Оператор аргументі → оператор нәтижесі

Және мұндағы X, Y, Z, W, V – бұл мынандай айнымалылар:

1. *Өту* (W) : (X, П, Y, Z) → (W, П, Y, Z)
2. *Қозғау* (V) : (X, П, X, Z) → (V, П, V, Z)
3. *Шығу* : (X, П, X, Z) → (X, Я, X, Z)
4. *Ұстап алу* : (ТБ, Я, ТБ , 0) → (ТБ, Я, ТБ , 1)



3.5.7-сурет. Маймыл туралы есептің күйлер кеңістігі

Егер есепті шешу үшін жоғарыда аталған $Т_О$, $Т_Я$, $Т_Б$, нүктелерінің мәндерін маңызды деп қабылдасақ, онда маймыл туралы есептің күйлер кеңістігінің бейнесін 3.5.7-суретіндегідей түрде көз алдымызға елестете аламыз. Бұл кенiстiкте тек 13 күй сипатталған, күй-графының доғалары қолданылатын оператордың тәртiптелген нөмiрiмен белгiленген. Кеңiстiкте маймылдың үш маңызды нүктелер арасындағы төрт сатылы жүрiсi (жәшiкпен немесе жәшiксiз) бейнеленген. Сонымен бiрге кеңiстiкте екi тұйық тармақ бар – ол маймылдың банан бумасы орналаспаған нүкте тұсында жәшiкке шығуы. Суреттегi қою сызықтар (бағытталған) төрт оператордан тұратын шешушi жолды бейнелейдi:

Өту ($Т_Я$); *Қозғау* ($Т_Б$); *Шығу*; *Ұстап алу*.

Қарастырылған мысал есептi тиiмдi түрде шешу үшiн таңдалып алынған күйлер түрiнiң қаншалықты маңызды екенiн көрсетедi. Мысалдағы көрсетiлген күйлер кеңiстiгi маймыл, жәшiк және банандардың ең алғашқы орналасқан үш нүктесiнен туады да, басқа нүктелер есепке алынбайды. Күйлер кеңiстiгiндегi iздеудi қысқарту үшiн көптеген алгоритмдер пайдаланылады. Осындай алгоритмдер түрлерiне *күйлер сызбанұсқасы* және *операторлар сызбанұсқасы* деп аталатын қуатты тәсiлдердi атауға болады. Оларда күйлер мен операторларды сипаттауға айнымалылар пайдаланылады. Күйлер сызбанұсқасында күйлердiң жиындары сипатталса, ал операторлар сызбанұсқасында белгiлi бiр типтегi iс-әрекеттердiң жиыны көрсетiлген. Бiз қарастырған мысалда операторлар сызбанұсқасы тәсiлi

пайдаланылған. Жоғарыда аталған екі тәсілдің осы есепке қатысты бейнеленуі келесі әдебиет көзінде кеңінен түсіндірілген [39].

Бақылау мысалы. Есептің қойылымы. Жоғарыдағы сипатталған «Маймыл әлемі» үнемі белгілі бір *күйде* болады. Күйлер белгілі бір уақыт аралығында өзгеріп тұруы мүмкін. Ағымдағы күй нысандардың өзара орналасуы бойынша анықталады. Бастапқы күйді табиғи тілдің мынандай сөйлемдері арқылы сипаттай аламыз:

1) Маймыл есік жанында
2) Маймыл еденде тұр
3) Жәшік терезе алдында
4) Маймылда банан жоқ.

Енді осы күйлерді есептегі нысандардың өзара байланысын көрсетуге ыңғайлы болуы үшін, келесі 3.5.1-кестедегідей бейнелейік.

3.5.1-кесте

Есеп нысандарының байланыстары

Күйлер	Нысандардың орналасуы			
бастапқы	Есік жанында	Еденде	Терезе жанында	Бар не жоқ
ағымдағы	Маймылдың жатықтағы орны	Маймылдың тік жағдайдағы орны	Жәшік орны	Маймылда банан бар не жоқ

Есепті бір ойыншыға арналған ойын есебінде қарастырайық. Осы ойын ережелерін былайша қалыптастырамыз.

1) Ойын мақсаты – төртінші бөлігі «бар болуы» деген мәнге ие күйге жету: күй (----, ----, ----, бар болуы).
2) Рұқсат етілген қандай жүрістер бар? Ондай жүрістің төрт түрі бар:
   (1) бананды ұстап алу
   (2) жәшікке шығу
   (3) жәшікті қозғалту
   (4) басқа жерге өту

жүріс (1-күй, М, 2-күй)

1-күй --------------------------- > 2-күй

М

1-күй – жүріске дейінгі күй.

2-күй – жүрістен кейінгі күй.

M – орындалатын жүріс.

Енді есепке қатысты барлық предикаттарды қарастырайық:

Жүріс (күй (ортасы, жәшікте, ортасы, жоқ болуы),
Ұстап алу,
күй (ортасы, жәшікте, ортасы, бар болуы)).
Жүріс (күй (P1, еденде, B, H),
өту (P1, P2),
күй (P2, еденде, B, H)).

Бұл сөйлемде мынандай тұжырымдар бар:

- Жүріс мақсаты маймыл P1 орнынан P2 орнына ауысады.
- Жүріске дейін де, орындалғаннан соң да маймыл еденде тұрады.
- Жәшік орны өзгермейді.
- Ал «қолында банан бар» күйі өзгермейді.

Есептегі ең маңызды мәселеге: «Маймыл бастапқы белгілі бір S күйінде тұрып, бананды ұстай ала ма?» сұрағын жатқызамыз. Бұл сұрақты да мынандай предикат түрінде: ***ұстай алады (S)*** өрнектеуге болады.

Мұндағы ***S*** – аргументі маймыл әлемінің күйі. Ал ***ұстай алады*** предикатының программасы екі түрлі бақылау нәтижесіне сүйене алады:

1) маймылдың қолында бананның бар болуы үшін, ***ұстай алады*** предикаты кез келген **S** күйі үшін **шын** болуы керек, бұл жағдайда, ешқандай жүріс қажеттігі болмайды. Предикат түрі:
   ұстай алады күй (----, ----, -----, бар болуы)).
2) басқа жағдайларда бір немесе одан да көп жүрістер қажет болады. Егер маймыл үшін **P1** күйінен **S2** күйіне ауысатын жүріс түрі бар болса, және сол күйде маймыл бананды ала алса (нөлден көп жүріс саны бар болса), онда ол бананды кез келген **S1** күйінде де ала алады.

Енді осы 1-дәрежедегі предикаттарды есептеу теориясындағы әдістерді пайдаланған осы «Маймыл мен банан» классикалық мысал есебінің Пролог тіліндегі кодасын келтірейік. Осы коданы пайдалана отырып, Турбо-Пролог ортасында аталған есепті шешу керек.

Пролог тіліндегі «Маймыл мен банан» мысалының кодалары.

```
predicates
    get_it.
    banana (symbol, symbol, symbol).
    search (symbol, symbol, symbol).
```

```
clauses
   goal
      write(«Enter coordinates of monkey...»), nl,
      readln(M), nl,
      write(«Enter coordinates of banana...»), nl,
      readln(B), nl,
      write(«Enter coordinates of chair...»), nl,
      readln(C), nl,
      search(M,B,C).
   search(O,A,H):-
      O=H,
      write(«Monkey near chair»), nl,
      banana(O,A,H);
      write(«Monkey is far away from chair. Please enter new position
for a monkey to move to...»), nl,
      readln(N),
      search(N,A,H).
   banana(X,Y,Z):-
      X=Y,
      write(«Monkey is under banana»),
      get_it;
      write(«Monkey is far away from banana. Enter new position for
a monkey and banana to move to»),
      readln(K),
      banana(K,Y,K).
   get_it:-
   write(«Monkey is under banana and can get it, Hallelua. Do you
want to take it?»),nl,
   readln(P),
  P=»yes»,
      write(«Oh, Yeah»);
      write(«I'm too young to die - said Monkey»).
```

Осы жаттығуларды орындау нәтижесінде мына терминдерді түсініп, оларды нақты есептерге қолдана білу қажет: ***предикаттарды есептеу, резольвента, термдер, атомдық формулалар, дұрыс құрылған формула (ДҚФ), резолюция принципі, жалпылық кванторы, бар болу кванторы, логикалық түрде шығару тұжырымдамасы, бос сөйлем, математикалық логика операциялары.***

Тапсырмалар:

1. Күйлері ДҚФ жиындарымен сипатталған компьютерлік ойын программасын құру.
2. Прологтан басқа тілде «Маймыл және банан» ойынының программасын құру.
3. Атауы «Қасқыр, ешкі және қырыққабат» деген классикалық мысалдың компьютерлік ойын программасын құру.
4. Атауы «Адам жегіштер мен миссионерлер» деген классикалық мысалдың компьютерлік ойын программасын құру.
5. Толық қарап шығу стратегиясын пайдаланатын компьютерлік ойын программасын құру.

3-бөлімнің бақылау сұрақтары:

1. ЖЗ-нің негізгі ұғымдары.
2. Күйлер кеңістігінде есепті шығару әдістері.
3. Зерделік жүйелердің негізгі қызметі.
4. ЖЗ-гі эвристикалық және алгоритмдік әдістер.
5. Толық караудың оңтайлы алгоритмі.
6. Толық қарау әдісінің сапасын бағалау белгісі.
7. Редукция әдісінің операторлары.
8. Зерделік жүйелердегі күйлерді сипаттайтын негізгі құрылымдар.
9. Редукция әдісіндегі жоспарлау механизмдері.
10. ЖЗ-дегі толық қарау әдісі.
11. Тереңнен іздеу әдісі.
12. ЖЗ-дегі төбелерді бағалау әдісі.
13. ЖӘНЕ/НЕМЕСЕ графының негізгі максаты
14. 1-ші дәрежедегі предикаттарды есептеу теориясының негізгі ұғымдары
15. ДҚФ (Дұрыс құрылған формула) анықтамасы, формуласы
16. ДҚФ құрудағы жалпылық және орындалу қасиеттері қандай?
17. Математикалық логикадағы квантор нені білдіреді?
18. Мына «~, =>» белгілер нені білдіреді?
19. ДҚФ тағы аксиома деген не?
20. Резольвенция ұстанымы.

4-бөлім. САРАПШЫ ЖҮЙЕЛЕР

Сарапшы жүйедегі программалардың жасанды зерде тараулардындағы басқа бағыттардан ерекшелейтін қасиеттеріне мыналарды жатқызуға болады: бұл жүйедегі программалар құрылымы қиын, іс жүзінде маңызы бар, шешім жолын табуы қиын есептерді шығаруға қолданылады, жүйе өзінің тапқан шешімі туралы түсінігін пайдаланушыға сапалы деңгейде түсіндіріп бере алуы керек. Бұл – оның сандармен жұмыс істейтін алгоритмдерден ерекшелігі. Табылған шешім өзінің сапасы мен тиімділігі жағынан адам тапқан шешімнен кем болмауы керек және маманмен пікір алысуы арасында өзіндегі бар білімді үнемі толықтырып отыратын қасиеті болуы керек. Осы қасиеттеріне байланысты сарапшы жүйе өзінің құрамында бірнеше бөліктер болуын қамтамасыз етуі керек. Ол бөліктерді атап айтсақ: білім қоры, жұмысшы зердесі, түсіндірме-басқару құралы, пайдаланушыға түсінікті болуын қамтамасыз ету бөлігі, білімді ала білу бөлігі, түсіндіру бөлігі. Осы бөліктердің ішіндегі соңғы үш бөлікті жүйе интерфейсіне, яғни жүйе мен адам арасындағы байланысты орындаушы бөліктер деп қарауға болады. Бұл қызмет табиғи тілді пайдаланумен тікелей байланысты.

Сарапшы жүйе түріндегі компьютер программалары пайдаланушы адаммен табиғи тілде сұхбат жүргізеді. Жасанды зерде теориясының зерттеулерінде табиғи тіл мәселесіне көп орын беріледі. Қазіргі кезде адамша сөйлей, түсіне алатын программалар жоқтың қасы. Дегенмен белгілі тақырыпқа қарапайым сөйлемдер құрылысын енгізу арқылы сұхбат жүргізе алатын компьютерлік программалар жасалған. СЖ программаларындағы пайдаланушы мен сұхбат жүргізетін программалар, әдетте, мынандай қасиеттерімен сипатталуы мүмкін: компьютер сұхбатты базалық (негізгі) мағлұматты жинақтау үшін жүргізеді де, өзінің білім қорындағы сипатталған жағдайлармен салыстырады. Яғни сұхбаттың сипаты мынадай болып келеді: кез келген түрде берілген қарапайым сөйлемдерден тұратын жауаптарды түсініп, қорыту қабілеті; білім қорындағы берілген жағдайларға сәйкес, программаның сұрақтар қоя білу қабілеті; қажет болғанда өзі келген қорытындысын табиғи тілде түсіндіре алу қабілеті.

Табиғи тілді түсіну мәселесін шешу үшін бірнеше сұраққа жауап табу керек. Біріншіден, адам білімінің үлкен ауқымды көлемі қажет. Табиғи тілді түсінуді іске асыратын компьютерлік жүйеде күрделі де нақты дүниедегі шым-шытырық байланыс түрлері сипатталуы

қажет. Екіншіден, табиғи тілдің өзіндегі құрылымдардың орналасуы белгілі бір тәртіпке бағынады. Бұл тәртіпті сақтамаса, тілді түсіну қиындайды. Табиғи тілдегі мағлұматтарды қамтитын сұрақтарды адам немесе компьютер тудырған өнім есебінде бағалауға болатындықтан, тілдің құрылысында оны да есепке алу қажеттігі бар.

Табиғи тілде сұхбат жүргізу әдістері. Табиғи тілде сұхбат жүргізуді мына кезеңдерге бөлуге болады: *грамматикалық талдау, түсіну* және *қорытынды жасау*, сонымен бірге синтез. Грамматикалық талдау мен синтез тілдер аумағындағы білімдер көмегімен, ал түсіну мен қорытынды жасау – ұғымдар аумағындағы білімдер көмегімен іске асады. Тіл аумағындағы сөйлем мағынасын түсіну былайша өтеді: сөйлемді грамматикалық талдау сәтті аяқталса, бірақ кейбір таңбалар тізбегін оқып-түсіну сәтсіз болса, онда «Қайталап анықтау» деп аталатын модуль іске қосылады. Сұхбат жүргізу нәтижесінде анық емес сөздер, сөйлемдер кездесуі мүмкін.

4.1. Білімдерді пайдалану үлгілері

Сарапшы жүйе дегеніміз – белгілі бір сала мәселелерін шешуде сарапшы-маман адамды ауыстыра алатын жасанды жүйе. Бұл жүйе жөніндегі зерттеулердің мақсаты – құрылымы қиын есептерді шығарғанда сапасы мен тиімділігі жағынан адам тапқан шешімнен кем түспейтін нәтижеге жететін компьютерлік программалар жасау. Яғни әңгіме сарапшы-адамның ақыл-ой еңбегін автоматтандыру жайында болмақ [27]. Сарапшы жүйедегі программалардың жасанды зерде тарауларындағы басқа бағыттардан ерекшелейтін қасиеттеріне мыналарды жатқызуға болады:

- бұл жүйедегі программалардың құрылымы қиын, іс жүзінде маңызы бар, шешім жолын табуы қиын есептерді шығаруға қолданылады;
- жүйе өзінің тапқан шешімі туралы түсінігін пайдаланушыға сапалы деңгейде түсіндіріп бере алуы керек. Бұл оның сандармен жұмыс істейтін алгоритмдерден ерекшелігі;
- табылған шешім өзінің сапасы мен тиімділігі жағынан адам тапқан шешімнен кем болмауы керек;
- маманмен пікір алысуы арасында өзіндегі бар білімді үнемі толықтырып отыратын қасиеті болуы керек.

Осы теориялық қағидаларға байланысты әдеттегі сарапшы жүйе өзінің құрамында бірнеше бөліктер болуын қамтамасыз етуі керек.

Ол бөліктерді атап айтсақ: білім қоры, жұмысшы зердесі (деректер қоры), түсіндірме-басқару құралы, пайдаланушыға түсінікті болуын қамтамасыз ету бөлігі, білімді ала білу бөлігі, түсіндіру бөлігі. Бұл бөліктер 4.1-суретте көрсетілген. Суретте келтірілген бөліктердің бәрінің бірдей сарапшы жүйе құрамында болуы міндетті емес, дегенмен де кейбір бөліктер жүйенің негізін құрады. Міндетті түрде болатын бөліктерге 1, 2, 3-ні жатқызуға болады.

Жұмыс істеуге қабілеті бар жүйе мына сипатта болады:

– ондағы мәліметтер мен білімдерді ұсына алатын әдіс болуы қажет;

– пайдаланатын мағлұматтар құрамы анықталған болуы керек;

– басқару құралы – интерпретатордың жұмыс істеу тәсілінің анық болуы.

4.1-сурет. Әдеттегі сараптамалық жүйенің құрылымы

Сараптамалық жүйені құрудағы негізгі мақсаттар қатарына мыналар жатады:

1) бір саладағы сарапшы-маман адамның құрылым жағынан қиындық келтіретін білімдерін жүйелеуге мүмкіндік беруі;
2) адам-сарапшы қабылдайтын шешімнің сапасын арттыру;
3) пайдаланушы адамды есептеуді көп қажет ететін қиын жұмыстардан босатуы сияқты мақсат – талаптар жатады.

Аталған мақсаттарға жету үшін, сараптамалық жүйе жұмысының екі түрлі тәртіпте орындалуы керек: ***білімді ала білу*** және ***есепті шығару тәртібі*** [30].

Білімнің деректерден өзгешелігін білдіретін бес ерекшелігі бар:

1. *Жіктелуі* (интерпретируемость). Бұл ерекшеліктер көз алдымызда бейнелеу үшін мынандай мысал келтірелік. Бізге бір мекемеде

жұмыс істейтін адамдар туралы мағлұматтар кесте түрінде берілді. Егер осы кестенің ішіндегі қос сызықпен қоршалған кішкене кестедегі мәліметтерді машина кодымен белгілеп, машина жадына салып қойсақ, онда бұл кестедегі сандар деректер жиыны болып табылады. Ал бізге «№ 26 іс қағазындағы бүкіл мағлұмат керек» деген өтініш түссе, онда осыған байланысты басқа мәліметтердің сақталу адресін анық білген соң ғана керекті программаны жаза аламыз. Егер машина жадында кішкене кестедегі жиыннан басқа, оның сыртындағы мағлұматтарды сақтасақ, онда осы кестедегі тік жол мен ұзын жолдар арасындағы байланыстар түрін анықтап, оны да машина жадында сақтай білсек, онда кестеге қатысты көптеген сұрақтарға жауапты машинадан алған болар едік. Сонымен, білімдердің мәліметтерден ерекшелігі – олар туралы машина жадында мағлұматтарды жинақтап сақтауға болады.

2. *Топтау қасиеті.* Мәліметтермен жұмыс істегенде, олардың бір жиынға кіру қасиетін анықтайтын байланыстар көрсетілмейді. Бұл бір түрдегі хабарларды бірнеше қайтара жазып, еске сақтауды қажет етеді, осы элементтердің барлығына қатысты іс-әрекетті бірнеше рет қолдануға мәжбүрлейді. Білімдермен жұмыс істегенде, бұл қиындықты жеңуге болады. Ол үшін білім элементтері арасындағы қарым-қатынас түрлерін анықтап, оны машинада сақтап отыру қажет.

3. *Жағдайды білдіру қасиеті.* Байланыстың бұл түрі машина жадын-дағы білімдердің бір-бірімен бірге бола алу-алмау қасиетін білдіреді. Бұл байланыстар бір мезгілде бір жерде болу, бір апаттық жағдайға тап болу, т.б. сияқты қасиеттерді білдіреді. Бұл қасиеттердің – *құрылымдық, функционалдық, каузальдық, семантикалық* деп аталатын түрлері болады. *Құрылымдық* қарым-қатынастар элементтер арасындағы бірінің ішіне бірі кіру қасиетін білдіреді. *Функционалдық* байланыстар элементтердің біреуін екіншісі арқылы есептеп табуға мүмкіндік береді. *Каузальдық* деп аталатын байланыстың түрі себеп-салдар бойынша анықталады, ал *семантикалық* түрге аталған үшеуінен басқа дүниедегі бүкіл байланыстардың бәрі кіреді. Білімдердің бұл ерекшелігі бір-бірімен қарама-қайшылықта болатын білімдерді іздеп табатын іс-әрекеттер құру мүмкіндігін береді.

4. *Семантикалық өлшемде болу қасиеті.* Хабарлар жинақталған жиындарда құрылымдардың бір-біріне жағдай бойынша жақындығын білдіретін байланыстар болады. Бұл байланыс білімдерді жағдайлар бірлігі бойынша топтауға мүмкіндік береді. Оның ерекшелігі – машинаға жаңа ұғым кіргізілген жағдайда, оны өз тобына орналастыру мүмкіндігін береді.

5. *Белсенділік қасиеті.* ЭЕМ пайда болғаннан бастап ондағы хабарлар – деректер, командалар деп екіге бөлініп, деректер үнемі баяу жағдайда, ал командалар белсенді болып келген. Машинада жүріп жатқан жұмыс барысының барлығы дерлік командалар әмірімен жүргізіледі де, мәліметтер керек уақытында ғана іске қосылады. Ал білімдермен жұмыс істейтін жүйелер үшін іс-әрекеттің белсенділігі сол жүйедегі бар сапалық ережелерге тікелей байланысты. Яғни программаның орындалуы-орындалмауы білім қорының сол мезеттегі күйіне байланысты.

Білімдердің осы аталған **бес қасиеті** оның мәліметтерден ерекшелігін сипаттайды. Бұл ерекшеліктерді шекара есебінде белгілесек, одан өткен мәліметтер білім болып саналады да, ал деректер қоры білім қорына ұласады.

Әрбір салаға қатысы бар кез келген білімдер екі түрде болады: *жалпы* және *жеке*. *Жалпы білімдер* – арнаулы әдебиеттер мен кітаптарда жазылып жинақталған, осы салаға байланысты анықтамалар, фактілер, ілімдер (теория). Ал саладағы мамандардың осы мәселеге қатысты өздерінің жеке білімдері де болады. Олар көбіне әдебиеттерде жазылмаған, жүйеленбеген. Сараптамалық жүйені құрудағы негізгі мақсат – осы маманның жеке білімін тауып, жинақтау. Көбінде бұл жеке білімдер эмпирикалық ережелерден тұрады. Оларды әдетте *эвристикалық* деп те атайды. *«Эвристика»* (грек сөзі, heuriska – табамын) дегеніміз – мәселенің көп шешімдері ішінен бір шешімді таңдап алуға кететін уақытты қысқартатын тәсілді табатын, сол шешімді іздеу, табу ісі барысындағы заңдылықтар мен тәсілдерін зерттейтін ғылым. Эвристикалар мамандарға мәселені шешуде ақылға қонымды ұсыныстар жасап, болашақта мәні белгісіз мәліметтермен жұмыс жасауға мүмкіндік береді [28].

Сараптамалық жүйе құрамы. Әдеттегі сараптамалық жүйе өзінің құрамында әртүрлі бөліктерді жинақтайды. Төменде осы бөліктердің атқаратын қызметіне шолу жасайық. *Білімдер қоры* – бұл жүйенің ең маңызды бөлігі. Жүйенің қуаттылығы осы білім қуаттылығына тікелей байланысты. Білімдерді жинақтайтын қорды құру аса көп еңбекті қажет етеді. Бұл қиындық қордағы білімдерді белгілі тәртіппен өзара орналастыру қажеттілігімен тікелей байланысты. Ал өзара орналастыру дәрежесі осы саладағы шығарылатын есептердің типі түріне, үлгісіне тікелей байланысты. Орналастыру тәртібі Қордан керекті мағлұматты тезірек таба білу үшін де үлкен рөл атқарады. Тезірек таба қою өнері іздеу стратегиясына байланысты. Жүйе дұрыс жұмыс істеуі үшін,

қордағы білімдер, мағлұматтар дұрыс екшеленіп, сұрыпталуы қажет. Сұрыптау ісі былайша өтеді: шешілетін саладағы фактілерді (ақиқат, белгілі нәрселер) мәселені шешу әдістерінен бөліп қарау қажет. Сонда фактілер – *мағлұмдамалық*, ал шешу әдістері – *процедуралық* деп аталады. Процедуралық мағлұматтарға өңделген білімдер мен логикалық қорытындылау әдістері кіреді. Бұлар жасалатын іс-әрекеттің түпкілікті мақсатының жүйелілігін бейнелеп түсіндіру қажет. Мағлұмдамалық мәліметтерге жататын білімдер сол саладағы фактілер мен аксиомалар жинағы болып табылады. Білім қорының тағы бір ерекшелігі – оның "сауаттылық" көрсеткіші болуы. Бұл көрсеткіштің белгісі – керек болған кезде, қажет білімдерді іске қоса алу мүмкіндігі. Білімдерді осыған орай ұйымдастыру үшін, деректердің бір-бірімен қалай байланысуы, білімдерді іздеудегі құралдар, салыстыру әдістері жөніндегі сұрақтарға жауап табу қажет. Деректер арасындағы байланыстарды екі түрге бөлуге болады: *ішкі, сыртқы*. Ішкі байланыстар ұғымдарды бір бөлікке топтап, олардың құрылымдық ерекшелігін береді. Сыртқы байланыстар осы топтардың арасындағы қарым-қатынасты білдіреді. Оның өзін *логикалық, ассоциативтік түрлер* деп бөлуге болады. Біріншісі – элементтер арасындағы байланысты білдірсе, екіншісі – оларды іздеуге арналған қарым-қатынас түрін береді. Осылардың бәрі қорда білімді пайдалану үлгілері арқылы сақталады. Білімді іздеудегі құралдың негізгі міндеті – жұмысшы зердесіндегі ұғымды тауып, оны қордағы мәліметпен салыстыру кезеңіне әкелу. Салыстырудың төрт түрі болады: *синтаксистік, параметрлік, семантикалық, міндеттелген*. *Синтаксистік* салыстыруда үлгілер бір-бірімен салыстырылады не мүлде салыстырылмайды. *Параметрлік* түрде салыстыру дәрежесін анықтайтын көрсеткіш енгізіледі, ал *семантикалық* түрде үлгінің өзі емес, олардың атқарылатын қызметі салыстырылады. Ал *міндеттелген* түрде салыстыруға түсетін үлгі басқа ұғым тұрғысынан қаралады. Қорды білімдермен толықтыру үшін, білімді пайдалану үлгілерін білу қажет. Білім қорының тағы бір тамаша қасиеттерінің қатарына оның өзіндегі білімдерді толықтырып отыру мүмкіндігі жатады. Ондай мүмкіндік жүйенің білім алу бөлігінде қаралады. Сонымен, білім қоры – бір саладағы маманның сұрыпталған, екшелген, белгілі бір тәртіппен ұйымдастырылған, жинақталған білімдер жиыны. Қордың көлемі, қуаты, ондағы білімді пайдалану үлгісі түріне байланысты. Қордың маңызды көрсеткіштерінің қатарына оның: «сауаттылық» көрсеткішін, білімдерді жаңалап толықтыруын, «түсініктілік» көрсеткішін жатқызуға болады. *Деректер қоры.* Бұл

қорды белгілі бір тәртіпппен жинақталған, көп адамдарға пайдалануға ыңғайлы хабарлар жиынтығы ретінде қарауға болады. Қордың негізгі міндеті – мәселеге қатысы бар мәліметтерді бір жерде сақтай білуі және қажет болған уақытта оларды пайдалану мүмкіндігін қамтамасыз ету. Бұл қорды кейде жұмысшы зердесі деп те атайды. Қорды толықтыру үшін мәліметтердің құрылымдық түрін анықтайтын жобалар қажет. Жобалауды іске асыратын мәліметтер дегеніміз – мәліметтердің құрылымдық түрін анықтайтын ережелер жиыны. Үлгілерді құрмас бұрын, мәліметтер табиғаты жайында бір-екі ауыз айта кету керек. Мәліметтер қарапайым не құрылымы қиын түрінде кездеседі. Егер мәліметті бір ұғым ретінде қарайтын болсақ, онда оның ішкі қасиеттерін білдіретін көрсеткіштері болады. Егер ұғымның өзін, оның қасиетін, қасиетінің сипатын, т.б. бір жерге жинақтасақ, онда осы ұғым туралы біраз мағлұмат жинауға болады. Қазіргі уақытта осындай үлгінің үш түрі кездеседі: *релятивтік, иерархиялық, торлық* түрлері. *Релятивтік* түрдегі үлгілерде ұғымдар арасындағы қарым-қатынастар кесте түрінде беріледі. Кестенің тік жолына ұғымдардың қасиетін білдіретін – *атрибут* деген көрсеткіш жазылады да, кестенің ұзын жолына олардың қабылдайтын мәні жазылады. Қабылдайтын мәндердің орналасу тәртібі аса ұқыптылықты қажет етеді, яғни жолдарын бір-бірімен ауыстыруға болмайды. Оларды әртүрлі кестелерде берілген ұғымдар арасындағы қарым-қатынасты «кілттер» деп аталатын байланыс арқылы беруге болады. Бұл байланыс түрі релятивтік үлгінің ерекшелігінің біріне жатады. *Торлық* үлгілердің негізінде тор ұғымы жатыр. Тордың төбесі ұғымдарды, ал оларды біріктіретін жолдар ұғымдар арасындағы байланысты білдіреді. *Иерархиялық* үлгідегі мәліметтер қатаң тәртіппен орналасып, оның құрылымдық түрі ағаш бұтақтарындағыдай болып келеді. *Интерпретатор – басқару құралы.* Әдеттегі программалау әдістеріне жасалған прорграммалар оған берілген ат бойынша іске қосылады. Сондықтан программалаушы адам мәселені шешудегі түрлі жолдарды ескеріп, оны басқару құралына кіргізе білуі қажет. Яғни программаның бір бөлігі өз жұмысын аяқтағанда, келесі кезекте қандай программа қандай мәліметтермен жұмыс істейді, соның бәрін анық көрсетуі керек. Басқарудың мұндай түрі құрылымдық жағынан қиын есептерді шығаруға жарамайды. *Пайдаланушыға түсініктілікті қамтамасыз ететін бөлік.* Адам мен жүйенің арасындағы қатынас екі түрде болады: сұхбат, сезімталдық терминал деп аталатын монитор. Біріншісінде – табиғи тілдің жүйеге салынған сөз қоры арқылы қатынас болады. Екіншісі програмалаудан

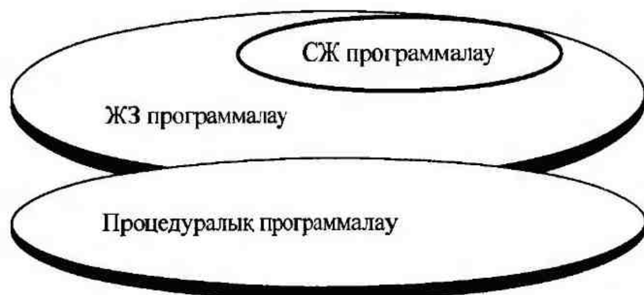
мүлде хабары жоқ адамдарға арналып жасалады. Бұл жағдайда пайдаланушы адам саусағының ұшын экранның белгілі бір жеріне тигізсе болады. Сонымен, пайдаланушы адамға түсініктілікті қамтамасыз ететін бөліктің негізгі міндеті – есепті шығару барысында «адам-жүйе» арасындағы сөйлесу қызметін қамтамасыз ету, оның нәтижесінде жүйеге түскен табиғи тілдегі сөз формаларын машинаға түсінікті түрге аудару. *Білімді алу білу бөлігі.* Сарапшы жүйе құру ісі барысында білімді ала білу өнері аса қажет, маңызды істің бірі болып табылады. Өйткені жүйенің негізгі мәйегі болып табылатын білім қорының сапасы осы білім алу ісіне байланысты. Жүйені құруда бұл бөлік өте қиын сатыға жатады. Бұл іс барысында мынандай қадамдардан өту қажет: 1) қолдағы бар білімді толықтыру қажеттілігі анықталады; 2) білімді алу барысы; 3) жаңа білім жүйесіне «түсінікті» түрге енуі керек; 4) жүйедегі білімдер өзгеріске ұшырайды да, кезек 1-ші қадамға беріледі. Осы айтылған қадамдарды адам не жүйе орындауына байланысты білімді ала білудің әртүрлі үлгілері бар. *Түсіндіру бөлігі.* Түсіндіру қажеттілігінің себебі: сарапшы жүйедегі программалар, құрылымдық түрі айқын емес есептерді шығаруға арналған. Яғни есептің шешімін табатын жолдарды анық көрсететін алгоритмдік әдіс жоққа тән. Сондықтан есеп шешімінің қалай табылғанын көрсете алатын құралдың болуы өте қажет. Түсіндіру қасиеті бар программаларды мынандай түрлерге бөлуге болады: Алдын ала дайындалып қойылған түсіндірмелер, релятивтік түсіндірмелер, арнайы түсіндіруші программалар. Бірінші түсіндіру түрінде табиғи тілде қойылған сұрақтарға жауап беру кезіндегі программаның іс-әрекетін есте сақтап қалуы. Бұл әдісті қойылатын бүкіл сұрақтарды қамтуға қолдануға болады. Бұл кезде пайдаланушы түсіндірмені компьютерде қалай жазылса, сол күйінде алуына болады. Релятивтік түсіндірмелерде программаның әр бөлігі өзінің іс-әрекеті жөнінде түсіндірмемен қамтамасыз етіледі. Арнайы түсіндіруші программалар жүйенің не істегінін бақылап және керек болған уақытта пайдаланушы адамға түсініктеме береді.

4.2. СЖ жобалау ерекшеліктері

Бұл параграфта сарапшы жүйелерді қолдану аймағына қатысты мәселелер қарастырылады. СЖ пайдалануды бағалау белгілеріне, оларды шектеу параметрлеріне көңіл бөлінеді. СЖ қолдану аумағы мен білімдерді пайдалану үлгілерінің түрлері бойынша жүйелеу зерттеледі. Қазіргі қолданыстағы СЖ шолулар жасалады.

Сарапшы жүйе – жасанды зерде программасының бір түрі. Жалпы жағдайда программалау тәсілін екі түрге бөліп карауға болады. Әдеттегі *процедуралық* программалау және *жасанды зерде әдістерімен* программалау.

Процедуралық програмалау тәсілінде компьютерге өте мұқият түрде есептің шешімін талдап, түсіндіріп беру қажет. Яғни бұл тәсілмен Бейсик, Фортран, Паскаль, және т.б. программалау тілдерінде программа жазылады [29]. (4.2-суретті қара).



4.2-сурет. Программалаудың үш түрі

Жасанды зерде әдістерімен программалау өзінің құрамында Сарапшы жүйе тәсілінде программалауды ұстайды. Программалаудың бұл стилінде беріліп, жинақталған алгоритмдік шешімі жоқ мәселелер қарастырылып, олар таңбалы механизм арқылы іске асатын шешу жолын пайдаланады. Ал ЖЗ программалары әдетте Лисп, Пролог сияқты таңбалы программалар тілдерінде жазылады. Мұндай программалар айнымалылары стек (виртуальды) компьютерінде болады, яғни компьютер жадының белгілі бір жерінде емес. Программалаудың бұл тәсілінде деректерді басқару үдерісі компьютердегі бар үлгілермен салыстыру және тізімдер құру арқылы іске асады. Тізімдерді құру технологиясы қарапайым болып келеді де, оның негізінде кез келген деректер құрылымын құруға болады [32]. ЖЗ программаларының әдеттегі программалаудан тағы бір ерекшелігі – ол оператордың мынадай жиынын пайдаланады:

Процедураны шақыру;
Бірінен соң бірін орындау;
Рекурсия.

Процедуралық программалардағы деректермен жұмыс істегенде, «=» немесе « :- » операторлары (мәнді беру) пайдаланылады Мысалы:

if-then-else – өту;
Do- while – процедураны шақыру;
repeat-until – кезегімен орындау.

Ал ЖЗ программалары әдетте *Лисп, Пролог* сияқты таңбалы программалар тілдерінде жазылады. Мұндай программалар айнымалылары стек сияқты виртуальды жадыда болады, яғни компьютер жадысының көрсетілген белгілі бір жерінде емес. Программалаудың бұл стилінде деректерді басқару **үдерісі** жадыдағы бар үлгілермен салыстыру және тізімдер механизмі арқылы іске асады. Тізімдерді құру механизмі қарапайым болып келеді де, ол бойынша кез келген айнымалыны өзіне сәйкес құрылыммен беруге болады.

Жасанды зерде тілдері мен процедуралық программалау тілдері арасындағы айырмашылықты мынадан көріп білуге болады: егер программа Фортран және Паскаль тілінде оңай жазылса, онда оны Лисп мен Прологта жазу қиын. Және керісінше, егер программаны Лисп пен Прологта жазу оңай болса, оны Фортран мен Паскальда жазу қиын [25, 26].

Енді ЖЗ мен СЖ программаларының арасындағы айырмашылықты анық сезіне білу үшін, төменде көпшіліккке кеңінен таныс ЖЗ программаларының тізімі келтірілген. Бұл программалар СЖ программасына жатпайды. Ол тізімге мыналар кіреді:

- кішкене (3 жасар) баланың айтқан әңгімесін қарап шығып, оны қайтадан әңгімелеп бере алатын программа. Бұл программа белгілі бір дәрежеде табиғи тілде «түсініп», оның себепсалдарлық байланыстарын анықтай білуі керек;
- адам дауысынан принтерге басып шығару. Пайдаланушы адам микрофонға сөйлейді, ал программа принтерге мәтінді басып шығарады;
- адам зердесінің жұмысын үлгілі көрсете алатын программа. Мысалы, геометриялқ фигуралардағы ұқсастықты табу есептері. Әдетте, мұндай есептер – *интеллектуалдық* (зерделік) *тестілер* деп аталатын топқа жатады;
- математикалық теоремаларды автоматты түрде дәлелдейтін немесе сол арқылы жаңа теорема аша алатын программалар;
- теледидар камерасы түсірген бейнелерге талдау жасай алатын программалар.

Ал СЖ программаларын жобалағанда оның негізі мақсаты мен құруын ерекшелеп, түсіне білу керек. СЖ құрудың негізгі мақсаты – мәселені маман адамдай шеше алатын компьютерлік программа құру, яғни зерделік түрдегі программалар құру [26].

СЖ программаларын бағалау белгілеріне мыналарды жатқызуға болады:

– программаның негізгі қызмет істеу әрекетін қайталай алатындығы;
– программаның өзінің тапқан шешімін адамға түсінікті тілмен түсіндіріп бере алатынындығы;
– программаның пайдаланушы адаммен табиғи тілде пікір алысу жүргізе алатындығы.

Сарапшы жүйе программалары жасанды зерде тәсілінде программалардың барлық жолдарын, әдістерін пайдалана алады. Мысалы, компьютермен шахмат ойнау программасын алайық. Бұл программалардың көпшілігінің негізінде «Дөрекі күш (грубая сызбанұсқа)» деп аталын тәсіл салынған, яғни ойын каталогында бүкіл мүмкін болатын ойын жолдарымен қатар, қарсы жақтың контр жолдары да салынады. Мұндай үлгіде бірнеше ойнау деңгейі қарастырылады. Бірақ барлық мүмкін болатын нұсқалардың өзара құрастыру саны өте үлкен болғандықтан, компьютер белгілі бір деңгейде ойынды тоқтатуға мәжбүр болады. Бірақ мұндай түрдегі программаны сарапшы жүйедегі программа деуге болмайды. Өйткені оның ішкі мазмұны зерделік емес, механикалық түрде болады.

4.3-сурет. Сарапшы жүйе архитектурасының негізі элементтері

Егер программа адамға тән белгілі бір эвристикалық іздеуді жасап, өз әрекетін адам тілінде түсіндіре алса, онда оны сарапшы жүйе программасына жатқызуға болар еді.

Сонымен, сарапшы жүйе программаларын құру архитектурасының негізгі элеметтеріне мыналарды жатқызамыз:

- *білімдерді ерекшелеу және басқару;*
- *іздеу тәсілдері;*
- *эвристика тәсілдері;*
- *ережелерге негізделген автоматты түрдегі талқылау;*
- *табиғи тілді пайдалану.*

Бұл элементтердің әрқайсысы жеке-жеке қарапайым болғанымен, олар барлығы біріккенде бірін-бірі күшейтіп, нәтижесінде қуатты компьютерлік программа бере алады. Сарапшы жүйе архитектурасының негізі элементтерінің топтасуын 4.3-суреттегідей көз алдымызға келтіре аламыз.

Іздеу тәсілдері. Жасанды зерде программаларында бастапқы күй, соңғы мақсатты күй сипаты болады, оны іздеу тәсілдері көрсетіледі. Жасанды зерде проблемалары әдетте кеңістіктегі күйі бойынша іздеу әдістерімен тікелей байланысты. Іздеу әдістері есептің мақсатын бір күйден екінші күйге өту арқылы іске асырады. Әрбір келесі қадамды жасау үшін, қарапайым механикалық стратегия пайдаланылады. Мұндай стратегия екі түрде болады: *тереңге іздеу, жайылып іздеу.* Егер таңдау механизмінде белгілі бір адамға тән тәсіл салынса, онда ол *эвристикалық іздеуге* жатады.

Эвристикалар. Егер шешім іздегенде белгілі бір формула немесе алгоритм болмаса, онда бір ережеге сүйеніп шешім ізделеді де, осы ережені *эвристика* деп атайды [31]. Мұндай эвристиканы пайдаланатын программаны жазу үшін адам-маманның ойлау механизмін зерттеу қажет. Ол үшін оны бақылау, интьервью жазу, статистикалық деректер жинау қажет. Көптеген эвристикалық программалар сол салаға қатысты адам-маман сарапшы ой тұжырымдарымен шешуге жетудегі тиімді жолдарды іске асырады. Яғни мұндай эвристиканы жүйелеп, белгілі бір қалыпқа келтіру үшін, сарапшы адамның іс-әрекетін зерттеу қажеттігі туады. Демек, адамның ойлау психологиясын зерттеп, оның ойлау деңгейіне белгілі бір түрде тәжірибелер жасау қажет.

Автоматты түрдегі талқылау. СЖ-дегі шешімді іздеудегі колданылатын тәсілдер әдетте ережелермен жұмыс істейді. Мысалы егер А шын болса, онда В-ны орында деген қарапайым продукцияның орындалуы арқылы іске асады. Бұл кез келген сала мәселесін осындай ережелер тізбегіне сыйдыра алсақ, онда автоматты түрде талқылау, орындау үдерісін іске асырған боламыз. Бұл үдерістегі негізгі ұғым – логикалық шығару. Ол екі аспектіде орындалады. Біріншісінде: бар

фактілер мен ережелерді белгілі бір жорамалды тексеріп, шын екенін нақтылауға пайдаланады. Оны тұжырымдаудың *тура тізбекті әдісі* деп атайды. Екіншісінде: жасалған тұжырымнан шын мәніндегі фактілерінің ережелерге шығу. Оны тұжырымдаудың *кері тізбекті әдісі* деп атайды.

Білімдерді ерекшелеу және басқару. Сарапшы жүйелерді жорамалдағанда, оның негізгі принциптерінің орындалуын қадағалау қажет. Осындай маңызды принциптің негізгісіне мынау жатады: салаға қатысты білімдерді сипаттайтын ережелер жиыны, шығару (іздеу) механизмінен бөлек құрылып сақталуы керек.

Табиғи тіл. Әдетте, сарапшы жүйе түріндегі компьютер программалары пайдаланушы адаммен табиғи тілде сұхбат жүргізеді. Жасанды зерде теориясының зерттеулерінде табиғи тіл мәселесіне көп орын беріледі. Қазіргі кезде адамша сөйлей алатын программалар аз деуге болады. Дегенмен белгілі тақырыпқа қарапайым сөйлемдер құрылымын енгізу арқылы сұхбат жүргізе алатын компьютерлік программалар жасалған.

Іске асырылған сарапшы жүйелерді шолу. Әдебиеттегі шолулар көрсеткендей, сарапшы жүйедегі программалардың жасанды зерде тарауларынан ерекшелейтін басты қасиеттерінің бірі – оның білімдер қоры болып табылады. Сондықтан іске асырылған программаларды жасау үшін, білімді ұсыну тілдері деп аталатын құралдар ойлап табылған. Қазіргі уақытта мұндай тілдердің үш түрі бар: *продукциялық, фреймдер* және *семантикалық торлар.* Енді осы үш түрдегі жасалған жүйелерге шолу жасаймыз [30].

Продукциялық ережелер түрінде жазылған жүйеге мысал ретінде MYCIN программасын келтіруге болады. Оның жұмыс істейтін саласы – медицина. Осы жүйеде қолданылған әдістер мен тәсілдерді пайдалана отырып, EMYCIN деп аталатын программа жасалды. Маманнан одан ары қарай да білімді ала біліп, білім қорын толықтыру үшін TEIRESIAS деп аталатын программалық құрал жасалған. Онда жүйедегі білімдердің өзі туралы мағлұматтар жинақталған. Оларды *мета-білімдер* деп атайды, яғни білімдер жайындағы мағлұматтар дегенді білдіреді. Бұл құралда мета-білімдер, мета-ережелер, шаблондар сияқты мағлұматтар іріктеліп, жинақталған және олар керек кезінде іске қосылып тұрады [32].

Білімді пайдаланудың әр түрлерінің жақсы жақтарын біріктіруге ұмтылудан туған үлгі есебінде *фрейм-үлгісін* атауға болады. *Фреймді* төбелер мен олардың байланыстыратын сызықтардан тұратын тор

есебінде көз алдымызға келтіруге болады. Әрбір *фреймнің* бір бөлігі осы фреймді қалай пайдалану туралы мағлұматтан тұрса, екіншісінде оны іске қосқанда қандай жағдайлар болуы туралы, үшіншісінде айтылған жағдайлар орындалмай қалғанда қандай амал қолдану керектігі жөніндегі мәлметтері сақталады.

Енді *семантикалық үлгідегі* жүйелерге келсек, бұл үлгілердің негізінде төбелері мен доғалары арқылы сипатталатын тор ұғымы жатыр. Тордың төбелері ұғымды білдірсе, оны қосып тұрған сызықтар олардың арасындағы байланысты білдіреді. Семантикалық тор үлгісіндегі әртүрлі байланыстар табиғи тіл сөйлемдердегі етістіктердің әр түрін өрнектеуі мүмкін.

4.3. Продукциялық типтегі үлгі

«Продукциялық жүйе» деп аталатын ұғымға қазіргі уақытта нағыз ғылыми түсініктеме беру қиын. Көбінде *продукциялық жүйе* деп **С** деген хабарлар құрылымына әсер ететін ұйымдастырылған немесе жекеше түрдегі программа жиындарының белгілі бір әдіспен ұйымдастырылған есептеу іс барысын айтамыз. Бұл программалар: Орындалу шарты **С** құрылымына деген кейбір талап-тілектерден тұрса, ал іс-әрекет осы талаптар орындалған жағдайда, істелуге тиісті жұмысты білдіреді [35].

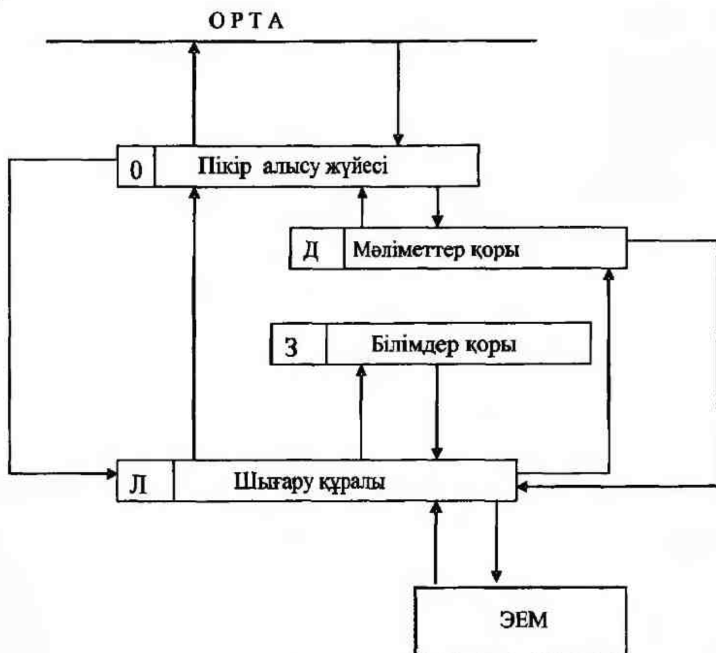
Жалпы жағдайда *продукция* деп мынадай өрнекті айтамыз:

(I); Q; P; $\mathbf{A_i \Rightarrow B_j}$**; N.**

Мұндағы **(I)** – продукция аты, **Q** – продукцияны қолдану аймағы, **P** – продукция ядросын қолдану дәрежесі, $\mathbf{A_i \Rightarrow B_j}$ – продукция ядросы, N – продукция аяқталғаннан кейінгі орындалуға тиісті жағдайлар.

Продукцияның негізгі элементі оның ядросы болып табылады. Продукция талдау секвенция белгісінің екі жағында тұрған $\mathbf{A_i}$ мен $\mathbf{B_j}$-ды қабылдайтын мәндеріне байланысты болады. Егер $\mathbf{A_i}$ «шын» мағынасында болса, онда $\mathbf{B_j}$-да «шын» мағынасын қабылдап іске қосылады. Продукцияның негізгі элементі болатын оның ядросын атқару қызметіне байланысты бірнеше топқа бөлуге болады. Ол үшін интеллектуалдық жүйенің әдеттегі сызбасын былайша көз алдымызға елестетуге болады (4.4-суретті қара). Мұндағы О, Д, Л, З – мен белгіленген интеллектуалдық программалардың әртүрлі мәліметтер, хабарлар, білімдер жиынын сақтап құрайтын бөліктері. Мұндағы О – интерфейс бөлігі, Д – дерекқор, Л – интерпретатор, З – білім қоры. Продукцияның ядросындағы $\mathbf{A_i}$ мен $\mathbf{B_j}$-дың осы суреттегі бөліктердің

қайсысынан өзіне арналған мәліметтер мен хабарларды алуына байланысты ядроны келесі 4.1- кестесіне келтірілген топтарға бөлуге болады [36].



4.4-сурет. Интеллектуалдық жүйенің әдеттегі сызбасы

Продукциялық ядро бөліктері мен интеллектуальдық программаның әртүрлі бөліктерінің қарым-қатынасын осы кестеден көруге болады. Егер **х** және **у** арқылы О, Д, Л, З бөліктеріндегі кез келген хабарды белгілесек, онда продукция ядросының Ах => Ву деген түрі А үшін хабардың **х** бөлігінен алынатынын білдіреді.

4.1-кесте

Продукциялық үлгі бөліктерінің қатынастары

X/Y	O	Д	З	Л
O		+		+
Д	+	+	+	+
З	+		+	+
Л	+		+	+

Кестедегі «+» белгісімен осы ядроның жиі кездесетін түрлері белгіленген. Мысалы, Аз, Вз ядроның екі жағындағы да мәліметтер білімдер қорынан алынады дегенді білдіреді. Ад, Вз А-дағы хабар мәліметтер қорынан, ал В-дағы білім қорынан алынатынын білдіреді.

Бұл үлгілердің кеңінен етек жайып, көп қолданыс табуына ерекше әсер ететін оның бірнеше қасиеті бар. Оларды атап өтсек: жалпы адамдағы білімдердің көбі топ-топқа бөліп, продукция түрінде қарауға лайықталған деуге болады. Кей продукцияны алып тастағаннан, не жаңаны қосқаннан олардың басқа топтарға әсері жоқ; егер қажет болған жағдайда мұнда кез келген алгоритм бойынша құрылған программаларды іске асыруға болады. Яғни ЭЕМ лайықталған білімдердің кез келген түрін продукцияларға сақтап, іске кірістіруге болады, онда кез келген саладағы білімдер жиының құрылымдық түрін сақтауға болады, ал оған қолданылатын іс-әрекеттер басқа жерде сақталуы мүмкін.

Осындай пайдалы, әрі қолдануға ыңғайлы қасиеттері бола тұра, продукциялық үлгілеудің өзіндік әлсіз жақтары да бар. Олар:

- әдеттегі программалау тәсілдеріне қарағанда, ондағы программалардың баяу жұмыс істеуі. Алайда бұл кемістікті аса жүйрік құралдарды қолдану арқылы толықтыруға болады;
- продукциялар саны өте көп болған жағдайда, оларды бақылап отыру қиындығы. Бірақ бұны да есептеу тәртібін өте қатаң бақылау арқылы жеңуге болады.

Продукциялар үлгісін қолдануға арналған катаң түрде эвристикалық жіктелген ілім саласы әлі жоқ. Мұнда көбінде эвристикалық ойлау белең алған. Мұндай ілім саласын құру қиындығы продукция ұғымының солқылдақтығында жатыр. Яғни продукция ядросының өзін әр саладағы мәселе ерекшелігіне байланысты қалай бұрсаң, солай соған жетектеледі, әрі продукцияларды іске қосу барысының өзі де сала мәселесінің ерекшелігі жетегінде жүре береді.

Продукциялық үлгідегі білімдерді іс жүзінде қолданудың тағы бір пайдасы – ол саладағы мәселенің статикалық жағдайынан динамикалыққа өтуін қамтамасыз етеді. Динамикалық жағдайға өтуге продукциялар тобында үнемі өзгеріс жасап тұру және кез келген уақытта іске қосылуға тиісті продукцияны динамикалық түрде бақылау мүмкіндігі оның тез, әрі сапалы орындалуына тікелей байланысты. Продукциялар тез орындалуы үшін, осы сала мәселесінің ерекшелігін ескеретін өте икемді басқару стратегиясын таңдай білу продукция үлгісін қолдану аясын одан сайын кеңейтеді [30].

Продукциялық ережелер түрінде жазылған жүйеге мысал ретінде MYCIN программасын келтіруге болады. Оның жұмыс істейтін саласы – медицина. Оның ішінде қанның инфекциялық ауруларына диагноз қою мен оны емдеу әдістерімен айналысады. Бұл жүйе жобаланып құрыларда оған бірнеше талап қойылған. Олар: маманның білімін бір жерге жинақтау қажеттігі, ескірген білімдерді алып тастап, оның орнына жаңа білімдерді жинақтайтын қабілеті болу қажеттілігі, ұсынған емі жайында түсіндірме бере алу қасиеті болуы. Ол тек дәрігер қолындағы құрал болуы, яғни ақырғы сөз адамда болуы керек. Осы аталған талап-тілектер жүйенің құрылымына, оған қолданған продукциялар әдісінің өте ыңғайлылығы, азғана уақытта білімдер қорын құрып, ұйымдастыруға мүмкіндік берді. Жүйеде мөлшермен 400-ге жуық ережелер бар, басқару стратегиясы әрекеттерге бойлау әдісімен жүргізіледі. Осы жүйеде қолданылған әдістер мен тәсілдерді пайдалана отырып, EMYCIN деп аталатын программа жасалды. Бұл құралды басқа сала есептеріне де қолдануға болады, өйткені оның білім қорын салаға байланысты білімдер жиынымен толтыру жеткілікті. MYCIN жүйесіне пайдаланушы адаммен пікір алысу ағылшын тілінде жүргізіледі. Ол үшін оның *«Лингвистический процессор»* деген программасында мәселеге қатысы бар көптеген ағылшын сөзінің қоры бар. Жүйенің тағы бір қасиеті – өзінің келген шешімін түсіндіріп бере алуы.

OPS-5 жүйесі. Бұл жүйе – продукциялық үлгідегі жүйелерді құруға лайықталған құрал түріне жатады. Ол АҚШ-тың Карнеги-Меллона университетінде жасалған. Бұл құралды білімдерді пайдалануға негізделген жүйелерді жасауға бағытталған арнайы программалық тіл деп те қарауға болады. **OPS-5** жұмысшы зердесі WME деп аталатын элементтерден тұрады. Ол элементтер мына түрде беріледі: [27].

<нысан типі, 1-атрибут аты, 1-атрибут мәні, 2-атрибут аты ... >

яғни WME-элементін «нысан-атрибут-мәні» үштігі түрінде қарауға болады. Мысалы <адам, Талғат, бойы 180см.> деген үштік бойы 180 см Талғат есімді адам бар екенін көрсетеді.

Бұл элементке сілтеме жасайтын ереже түрі мынадай болады:

(P – продукция аты;
(WMEc немесе – WMEc /теріс мәнде/) => (WME_A құру)).

Бұл продукцияның сол жағы – *анецедент*, ал оң жағы – *консеквент* деп аталады. Бұл ереженің орындалу мақсаты мынадай: егер

WMEc барлық қатары жұмысшы зердесінде болса, онда ол WME_A элементімен толықтырылады. Мысалы, сөйлеммен берілген мынадай ереже бар делік: «Егер жоба құрудың негізгі мақсаты сарапшы жүйе құру болса және оны құру құралына супер-мини компьютердің ақпараттық құралдары мен программалық қамтамасына **OPS-5** жатса, онда оны жасау құралы ретінде **OPS-5** жүйесін алуға болады».

Енді осы ережені продукция түрінде жазайық:

(Р – құрал табу;
(жоба ^ мақсаты- СЖ-құру);
(ортасы^супермини ақпараттық құралдары және OPS-5 программалық қамтамасы) → (құру ^ OPS-5 жасау ортасы).

OPS-5 жүйесіндегі ережелерде көрсетілетін WME элементін сипаттағанда, оның нақты мәнінің орнына айнымалыны көрсетуге болады. Мұндай айнымалылар < > жақшасымен белгіленеді. Мысалы, жоғарыда келтірілген ереже былайша жазылады:

(Р – құрал табу- Х ;
(жоба ^ мақсаты <Х>) ;
(программалық қамтамасы ^Аты <Y>^ мақсаты <Х>) ;
(құру ^ жасау құралы <Y>).

Бұл продукция сөйлем арқылы былайша бейнеленеді: «Егер жоба мақсаты <Х> болса және осы мақсатқа жету үшін <Y> деген программалық қамтама болса, онда оны құру ортасы есебінде <Y> программалық қамтаманы пайдалану керек».

Егер Х есебінде – сандық талдау, Y есебінде Фортран құралын алсақ, онда жоғарыда аталған продукцияны былайша жазамыз:

(Р- құрал табу;
(жоба ^ мақсаты – сандық талдау);
(программалық қамтамасы ^Аты Фортран мақсаты – сандық талдау) → (құру ^ жасау құралы Фортран).

Жалпы **OPS-5** – арнайы жүйе. Ондағы шешімдерді табу үшін RETE деп аталатын алгоритм қолданылған. Бұл алгоритм өзара дау тудыратын ережелерді бір жерге жинақтап, оны шешуге тырысады. Оны шешуге арналған тәсілдердің ішінен **OPS-5** жүйесі, LEX деп аталатын

тәсіл түрін пайдаланады. Бұл тәсілде ең соңғы қалыптасқан WME элементі бар ережелерге бірінші мән беріледі. Осындай ережелердің бірнешеуі жинақталса, онда оның ішіндегі сол жағында шарттар саны ең көп ереже бірінші орындалады. Сонымен **OPS-5** жүйесін сарапшы жүйені құруға арналған білім инженериясы тілдері қатарына жатқызады.

4.4. Фреймдік және семантикалық желі үлгілері

Бұл параграфта Білімдерді пайдаланудың фреймдік түрі, олардың типтері, Семантикалық желілер деп аталатын үлгі түрлеріне қатысты мәселелер қарастырылады.

Фреймдер. «Фрейм» терминінің анықтамасының өзі осы уақытқа дейін әртүрлі талқылауда болып келді. Бұл терминді алғаш рет 1972 жылы М.Минский кіргізген [34]. *Фрейм* деп кез келген ұғымның, нысанның, нәрсенің қасиетін сипаттайтын көрсеткіштер жиынын атаймыз. Егер бұл жиыннан олардың бірін алып тастасақ, онда осы нәрсе туралы ұғым туа алмайды (адамның ойында, машина жадысында). Фреймдік үлгіні ғылыми білімдерді бейнелеу үшін пайдаланған ыңғайлы. Оларды жинақтау, білімдерді автоматтық программалық жүйелерде сақтауға мүмкіндік береді. Фреймдік үлгі немесе М.Минскийдің фреймдік теориясына негізделген білімдерді пайдалану/ұсыну үлгісі адамның жадысы (немесе есі) мен оның санасының психологиялық үлгісінін мысалы бола алады деп есептеледі. Яғни бұл жүйеленген, бір қалыпқа келтірілген теориялық үлгі бола алады. Бұл теориядағы ең маңызды ұғымға *Фрейм* ұғымы жатады. *Фрейм* – кез келген тұжырымдық нысанды көрсете алатын деректер құрылымы. Фрейм теориясы – біздің көріп, естіп жүрген заттарымыздың түсінетін ұғымдарға қатысты психологиялық зерттеулерді бойына жинақтаған теория. Соның негізінде қабылдап алу тәсілдері бір көзқарас тұрғысынан түсіндіріліп, соның негізінде тұжырымдық үлгілеу іске асады. Фрейм теориясының негізінде адамның сырттан алған ақпаратты, нақты элементтер мен олардың мәндерімен салыстыру арқылы фактілерді жинақтау тәсілі жатыр. Яғни адамның жадында әр тұжырымдық нысан үшін белгілі бір қалып (рамка) – *шептер* анықталған. Фреймдер теориясын, шешім алатын теорияға қарағанда, мәселені қою теориясына жатқызған дұрыс. Бұл теорияны пайдалану – адам жадындағы болатын түсіну, қорыту, үйрену мәселесін көрсетумен бірге, ондағы

бар ұғымдар деңгейлерінің аражігін ашып, адам жадындағы болатын ойлау механизмін тәптіштеп қарау мүмкіндігін береді.

Фрейм теориясының негізгі ой арқауы мынада: адам жаңа жағдайға душар болғанда өз жадынан осы жағдайға қатысты белгілі бір құрылымды алып шығады. Мұндай құрылым түрін М.Минский *Фрейм* деп атаған. Сонымен, *Фрейм дегеніміз* – білімді пайдалану бірлігі, ол адам есінде бұрыннан сақталады. Ағымдағы жағдай өзгерген кезде, қажет болғанда, оның бөлшектері өзгеріп тұруы мүмкін. Әр *Фрейм* әртүрлі ақпаратпен толықтырылып тұруы мүмкін. Бұл ақпарат есебінде аталған *Фреймді* пайдалану тәсілдері қолдану нәтижесінде болатын мәліметтерді алуға болады. Әр *фреймді* бірнеше төбе мен олардың қарым-қатынасынан тұратын тор есебінде де карауға болады. Тордың ең жоғарғы деңгейінде тұрақты қалыптасқан ақпарат болады: ол – нысан күйін білдіретін және сол сәтте «шын» мағынасындағы мағлұмат. Келесі деңгейлерде терминалды слот деп аталатын терминалдар жиыны орналасады. Бұл слоттар нақты деректер және нақты мәндермен толықтырылып отыруы керек. Әр слотта міндетті түрде шарттар болады. Бұл шарттар атрибуттарға берілген белгілі бір мәндер, пайдаланушы ұсынған мәндер арқылы салыстырылып отырылады. Егер мәндер өзара сәйкес келсе, онда шарт орындалады. Шарттар *қарапайым* және *күрделі* болады. *Қарапайым* шарттар – белгі таңба (метка) арқылы беріледі. Ол таңбаларда мынандай талаптар болуы мүмкін:

а) сәйкестікті пайдаланушы қою керек;

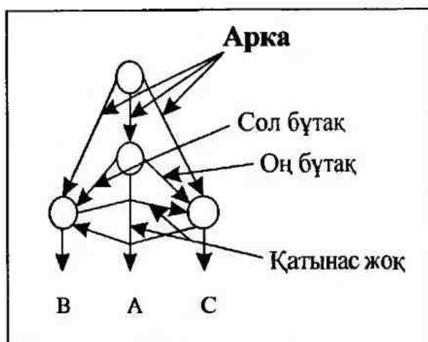
ә) мәндердің толық сипатталуы;

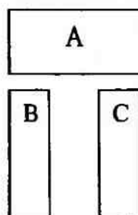
б) *Фреймнің* арнайы құрамдарының көрсеткіштерінің болуы.

Күрделі шарттар фактілер арасындағы қарым-қатынастарды көрсетеді. Бұл фактілер бірнеше терминалға сәйкес келуі мүмкін.

Негізгі мағынасы қатынастар болып табылатын фреймдерді жинақтасақ, біздер фреймдер жүйесі деп аталатын құрылымды аламыз [37].

Мысалы (4.5-суретті қара), «Арка» ұғымының фреймдік түрін қарастырайық. A, B, C бөлшектерінің өзгеруі де Фрейм бола алады. Арка ұғымының Фреймі A, B, C слоттарынан ғана емес, олардың арасындағы байланыстар арқылы да толықтырылады. Аталған жағдайды қарапайым шарт түріне жатқызуға болады. Бұл суретте «Арка» ұғымынын түрі әртүрлі жақтан қарауға байланысты толықтырылған. Екінші деңгейлі фреймдер әртүрлі жақтан қарағандағы арка түрі.





4.5-сурет. «Арка» ұғымының фреймдік түрі

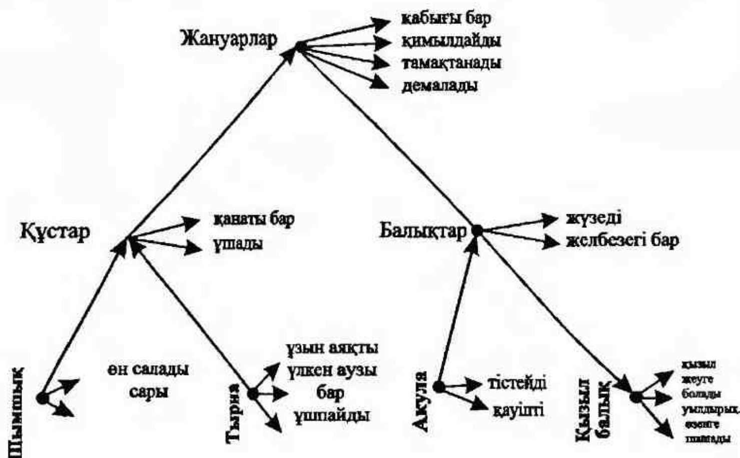
Фреймдердің бірнеше негізгі қасиеті бар [25]. Оларға мыналарды жатқызамыз:

Базалық тип. Фреймнің әртүрлі позициядағы мәліметтерін еске сақтау үшін көп жады көлемі қажет. Сондықтан аталған заттың ең маңызды нысандары базалық фреймдер түрінде еске сақталады. Соның негізінде жаңа күйлерге арналған фреймдер құрылады. Бұл жағдайда әр *фрейм* өз құрамында барлық фреймдерге ортақ бөліктерді пайдалануға мүмкіндік беретін нұсқағыштары бар слоттарды ұстап тұрады. Бұл – фреймдерге қатысы бар ортақ бөліктерді сақтайтын құрылым. Ол фреймдерге қатысты сыртқы мәліметтер өзгергенмен, өз қалпында сақталады. Мысалы, «Арка» ұғымындағы А (көлденең деталі), В (тік деталі), С (тік деталі) – осы детальдар мен олардың арасындағы байланыс. Осы үш ұғым – «Арка» фреймінің негізгі базалық слоттары. Осындай құрылымның қасиетіне әр кезде алынған мәліметтерді кеңінен пайдалануға мүмкіндік береді.

Сәйкестендіру, салыстыру үдерісі. Бұл үдеріс нәтижесінде *фреймнің* дұрыс таңдалып алынғаны тексеріледі. Ол үшін ең алдымен Базалық *фрейм* таңдап алынады. *Фреймде* слот мәнін шектейтін шарттар болады. Пайдаланушы анықтайтын мақсатты функция осы шарттардың қайсысы нақты кезеңдегі жағдайға (мәліметке) сәйкес келетіндігін анықтайды, яғни релеванттығы тексеріледі. Аталған *фрейм* өзінің релеванттығының дұрыстығын не бұрыстығын өзі дәлелдейді. Егер сәйкес келсе, салыстыру үдерісі аяқталады. Егер аталған *фреймде* қатесі бар слот анықталса, онда қосымша мәлімет қажет болады. Мысалы, «Арка» *фреймін* екінші тұрғыдан қарау (бүйірінен) *фрейм* слоттары мынандай: «деталь А-бар, В-бар, С-жоқ» деген қосымша мәлімет

береді. Яғни «С мен А байланысы бар, В мен А байланысы бар, В мен С байланысы жоқ» деген, онда басқару одан жоғары фреймге беріліп, фрейм анықталады, егер бұл мәліметтер сәйкес келмесе, онда басқару басқа фреймге беріледі, т.б.

Иерархиялық құрылым. Әдетте, фреймдердің құрылымы иерархиялық түрде сипатталып беріледі. Мұндай иерархиялық құрылымдардың ерекшелігі мынада: ең жоғарғы деңгейдегі фреймге қатысты атрибуттардағы ақпаратты осы фреймге кіретін төменгі деңгейдегі басқа фреймдердің бәрі пайдалана алады.



4.6-сурет. Иерархиялық құрылымның тұжырымдық сызбанұсқасы

Мысалы, иерархиялық құрылымы бар «Жануарлар» фреймінің сызбанұсқасын келесі 4.6-суреттен көруге болады. Осы «жануарлар» фреймінің слоттары одан төменгі деңгейдегі «балық» және «құс» фрейміне де, одан кейінгі деңгейдегі «акула» және «шымшық» фрейміне де жарай береді. Мұндағы «Қимылдайды» деген атрибут барлық деңгейдегі фреймдерге қатысы бар, яғни «Құстар», «Балықтар», «Шымшық», «Акула» сияқты фреймдерге қатысы бар.

Фреймаралық желілер. Классификациялық иерархиялық құрылымы бар тұжырымдық нысандарды іздегенде іздеу үдерісі сәтсіз аяқталса, онда осыған ұқсас тағы басқа фреймді іздеу қажеттігі туады. Бір-бірінен бөлектегіш нұсқағыштары арқылы байланысқан *фреймдер желісін* құра алады. Егер, мысалы, «стол», «орындық» деген ұғымдар фреймдерін бөлектегіш нұсқағыштармен байланыстырсақ, *фреймдер*

желісін аламыз. «Стол» деген фреймнің бөлектегіш нұсқағышына «арқалығы жоқ», «тамақ ішуге, жазу, оқуға арналған» деген ұғымдар жатса, «Орындық» фреймінің бөлектегіш нұсқағышына «столдан аласа», «отыруға арналған» деген ұғымдар жатады.

«Үнсіз келісім бойынша мәні болу». Фреймнің бұл қасиетіне қатысты бір мысал қарастырайық, мынандай мәтін бар делік: *«Айгүл Ержанның туған күніне шақырылды. Айгүл Ержанның шахмат ұнататынын біледі. Айгүл бөлмесіне кіріп, кішкене темір сандықшасын сілкіледі. Сандықшадан ешқандай дыбыс шықпады».* Бұл мәтіннен адам баласы, әрине, Ержанға сыйлық беру мәселесі айтылғанын түсінеді. Яғни мәтін сөйлемдері көмегімен толық мазмұнды қалпына келтіреді. Енді бұл сөйлемді ЕЭМ-түсіну үшін мынандай Фрейм құрамыз:

Туған күн → ***сыйлық*** → шахматтар → ***шахмат жоқ*** → ***сатып алу*** → ақша → сандықша → сілкілеу → дыбыс жоқ → ***ақша жоқ*** → ***атасынан сұрау***.

Бұл фреймдегі курсивпен белгіленген ұғымдарды *фрейм* есебінде компьютер жадында сақтау қажет, яғни бұл фреймдер үнсіз келісім бойынша фрейм мәні бола алады.

«Абстракты – нақты», «Бүтін – бөлігі» қатынастары. Бұл қатынастар иерархиялық құрылымдарда жиі кездеседі. Мысалы, «Жануарлар» фреймiндегі жоғарғы деңгейде абстракты нысандар, төменгі деңгейде нақты нысандар орналасқан. Әдетте, оларды байланыстыратын бұл қатынас түрін **IS-A** түріндегі байланыс дейді. Мысалы, «Акула IS-A балық». Екінші қатынас түрі – «Бүтін – бөлігі» деп аталатын байланыс. Бұл қатынас түрінде төменгі деңгейдегі ұғым жоғарғы деңгейдің бір бөлігі болып табылады. Мысалы, «Аудитория» және «оның қабырғасы» ұғымдары. Бірінші ұғым бүтінді, екінші оның бөлігін анықтайды. Бұл байланыс түрін әдебиетте **PART-OF** деп белгілейді.

Сонымен, М.Минский теориясына негізделген *фреймдер* сарапшы жүйелер программаларындағы білімдерді пайдалану үлгілерінде кеңінен қолдануға болатын құрылым екенін көрдік. *Фреймдік үлгілер* сарапшы жүйелерді құруда ғана емес, бейнені тану, табиғи тілді түсіну, білімдерді өңдеу сияқты көптеген жүйелерде пайдалануға болатын өте ыңғайлы құрылым болып табылады.

Торлық үлгілер. Білімді пайдаланудағы логикалық үлгілерде білімнің ұғымдары арасындағы қарым-қатынастар, формальдық жүйенің негізін құрайтын емлелік ережелер арқылы бейнеленеді. Білімге қатысты құрылымдарды анықтаудағы келесі қадам ретінде, енді осы қарым-қатынастардың түпкі мән-мағынасын анықтайтын

құрылымдары бар үлгілер тобы зерттеле бастады. Бұл үлгілер тобының негізінде төбелері ұғымдарды білдіретін, ал оларды қосып тұрған сызықтар – доғалар, олардың арасындағы байланыстарды білдіретін тор ұғымы жатыр. Төбелер мен доғаларды бейнелеп түсіндіруге белгілі бір шектер қоя отырып, тордың әртүрлі түрін алуға болады. Егер торлар төбелеріндегі ұғымдар құрылым жағынан қарапайым болса, онда оларды – *қарапайым*, ал ұғымдардың ішкі құрылым күрделі түрде кездессе, онда оларды *иерархиялық* түрдегі торлар деп атайды. Төбелер арасындағы байланыстар бір түрде болуы мүмкін, мұндай жағдайда торлар *біркелкі* деп аталады. Егер төбелер арасындағы байланыс түрлері әртүрлі мағынада болып келсе, торлар *әркелкі* деп аталады. Торлық құрылыммен берілген үлгілердің төрт түрін атап өтуге болады. Оған *функционалдық, сценарийлер, семантикалық торлар, фреймдер* жатады [26].

Функционалдық торлар. Оларда егер екі төбе доға арқылы жалғанса, доға бағыты бірінші төбеден екіншіге қарай болса, онда біріншісі екіншісінің аргументі болып табылады. Екінші төбедегі бейнеленген іс-әрекетті, шешімді табуға арналған жолдар деп есептеуге болады. Ал осы төбеден шығатын бағытталған доғаларды функцияның мәні деп есептейміз.

Сценарийлер. Бұл *біркелкі* торлар түріне жатады. Мұндағы байланыстар мағынасы әртүрлі болуы мүмкін. Көбінде бұл байланыстар – болуға тиісті жағдайлардың бірінен кейін бірі орындалатын іс-әрекет тізбелері. Бұл торлар байланыстардың әр түрлері ғана емес, төбелердегі ұғымдар мағынасының да әр түрде кездесуі мүмкін. Яғни семантикалық торларды класс деп есептесек, оның элементі есебінде функционалдық торлар мен сценарийлердің әр түрін алуға болады. Жалпы жағдайда семантикалық торлар үлгісіндегі жинақталған теориядан гөрі, сол шығарылып отырылған салаға ғана тән есептің ерекшелігін көбірек ескерген эвристикалық шешу үлгілері басым болып келеді. Бұл осы торлар үлгісінің негізгі кемшілігі болумен қатар, негізгі ең жақсы қасиетіне де жатады.

Семантикалық торлар түрінде үлгілеу. Білімдерді көрсетіп, пайдалану тәсілдерінің біріне *семантикалық тор* түріндегі үлгіні қарауға болады. Мұндай үлгі түрі психологияда «ұзақ уақыттық жады» деп аталатын құрылымды сипаттайтын тәсіл түрінде белгілі. Бұл тәсіл кейіннен білімді пайдалану үлгілерінің негізгі түрлеріне енді. «Семантика» деп сөздің, сөйлемнің, шығарманың іс-әрекеттің, жағдайлардың, мән-мағынасын анықтайтын ұғымды айтамыз. Басқаша

айтқанда, «Семантика» деп таңбалар мен осы таңбалар жиынтығынан тұратын нысандар арасындағы белгілі бір қарым-қатынасты айтуға болады. Ал «Прагматика» дегеніміз – сөз бен таңба құрушылары арасында болатын айшықты қарым-қатынас түрі.

Куиллиан үлгісінде тұжырымдық нысандар ассоциативтік торлар түрінде беріледі. Тор өз төбесіндегі ұғымдар, оларды байланыстыратын доғалар-концептер арасындағы қатынастарды білдіреді [35].

Куиллиан үлгісіндегі деректер фактілерге негізделген, ал оларды мынандай құрылым түрімен бере аламыз: элемент, қасиеті, нұсқағышы.

Элемент – факт, нысан, оқиға, ұғым немесе жеке сөз, зат есім, сөйлем, мысалы, «ит», «Африка», «әдемі гүл», «жақсы адам», т.б.

Қасиеті – элементті сипаттайтын құрылым. Ол сын есіммен, етістікпен, есімшемен берілуі мүмкін. Мысалы, «Ызалы», «Солтүстік», «Гүлді ұнатады», «Адаммен сөйлеседі», т.б.

Нұсқағыштар – элемент пен оның қасиетін байланыстыратын қатынастар. Мысалы, «Ызалы ит», «адамды» «қауып алды» деген етістік пайдаланылады.

Семантикалық тор түріндегі үлгі құру үшін, Куиллиан мен басқа да авторлар, саладағы ұғымдар арасындағы функцияларды былайша анықтауды ұсынған:

- Ішкі жиын – аға жиын.
- Индекстеу (сын есім, есімше).
- «Және», «немесе» деп аталатын логикалық ережелердің өзара топтау операторлары есебінде болу «жақын болу», «бірінен бірін шығару», «алдында болу», «ұқсас» деп аталатын катынастарды пайдалану ұсынылады.

Куиллиан құрған семантикалық тор үлгісінің маңыздылығы мынада:

- әдетте, қолданылатын құрылымды талдайтын семантикалық өңдеу әдістерінен тыс, ұзақ уақыттық жады құрылымы ұсынылады. Бұл құрылымда тілдік белсенділік көлеміне көп мән беріледі;
- фактілер мен ұғымдар арасындағы қатынастарды сипаттайтын тәсіл ұсынылады. Бұл тәсіл – семантикалық торлар деп аталады.

Мұндай семантикалық өңдеу тәсілі, *Мир понятий – ұғымдар әлемі* деп аталатын аумақта іске асады. Бұл *ұғымдар әлемі* – өзінің бойында белгілі бір алфавитті, ұғымдарды, ережелерді аксиомаларды жинақтаған нақты формальды теория. Семантикалық торлардың білімді пайдалану үлгілерінің басқасынан негізгі ерекшелігі – ол

жүйенің бүтіндігін қажет етеді. Яғни білім қорындағы білімдер мен оны шығару механизмі бір-бірінен бөлініп қаралмайды.

4.5. СЖ құрудың кезеңдері мен құралдары

Сарапшы жүйеге арналған құралдар. Әдеттегі программалау түрінде жоғарыда аталған белгілер бойынша тілді таңдау – өте қиын мәселе. Ал *сарапшы жүйе* құруға арналған таңдаулардың да өзіндік ерекшелігі бар. Зерттеулер көрсеткендей, *сарапшы жүйе* құру құралдарын мынадай төрт топқа бөлуге болады: *программалаудың таңбалық тілдері, білім инженериясы тілдері, сарапшы жүйелерді жобалауды автоматтандыру жүйелері, қабықтар.* Төменде осылардың әрқайсысына жеке тоқталамыз. *Программалаудың таңбалық тілдері.* Бұл тілдер сарапшы жүйелерді құру барысын тездетуге арналған. Бұл тілдер қатарына: таңбалы хабарларды өңдеуге арналған тіл – *ЛИСП* жатады. Бұл тілдерде жасалған құралдардың жақсы да, әлсіз де жақтары бар. Программалаудың іс-әрекет тілдері таңбалы хабар, мәліметтермен жұмыс істеуге өте ыңғайлы болғанымен, программалау тілдерінің төменгі дәрежесіне жатады. Яғни программалаушы адамға бірталай қиындық келтіреді [27].

Білім инженериясы тілдері. Бұл топқа программалаудың қисынды жүйедегі тілі – ПРОЛОГ және продукциялық түрдегі жүйелерді құруға арналған OPS-5 құралы жатады. ПРОЛОГ тілі сарапшы жүйелерді құруда едәуір табыстарға жеткізеді, дегенмен бұл тілдегі басқару әдісі тілдің құрамына кіргендіктен, жүйе бөліктерінің бір-бірімен қарым-қатынаста болуын біршама қиындатады. Жоғарғы дәрежедегі тілдер жүйелер құруда ең жақсы құрал болғанымен, оның уақытын өте ұзартып жібереді. *Пролог.* Сарапшы жүйе – программалау бағытындағы кеңінен қолданылатын тілдер қатарына Пролог тілін жатқызуға болады. Бұл, басқалармен салыстырғанда, – жас тіл, оның негізінде математикалық қисын әдістері жатыр. Бұл дербес компьютерлерге арналған сарапшы жүйелер, білім қоры, табиғи тілмен жұмыс істейтін жүйелер сияқты программалар құруға мүмкіндік береді. Пролог тілі жоғарғы деңгей тіліне жатады. Оның қатаң түрде жүйеленген теориялық негіздемесі бар. Ол математикалық логиканың тұжырымдамасы мен әдістерін пайдалануға бағытталған. Яғни Пролог логика терминдерінде программалауға арналған. Прологтың басқа тілдерден ерекшелігі – онда жазылған программалардың мағлұмдамалық сипатта болуы. Программаның негізгі құрылымдық блоктарына белгілі бір құрылымдағы нысандар жиыны және осы

байланыстыратын функциялар мен қатынастар жатады. Прологта әдеттегі программаларда кездесетін шартты операторлар, цикл немесе өту операторлары сияқты басқару конструкциялары болмайды. Пролог программасы шығаратын мәселенің үлгісі есебінде болады. Пролог программалау, ойлау стилінің басқа түрін қажет етеді. Сондықтан кейде мұндай жұмыс психологиялық белгілі бір күш салуды қажет етеді. Яғни әдеттегідей мәселені шешу үшін, белгілі бір іс-әрекет тізбегін берудің орнына, Пролог программасында нысандар мен олардың қарым-қатынасы терминдері арқылы есептің мазмұнын беру қажет. Программалаушы Прологта есеп алгоритмінің орнына, оның логикалық спецификациясын құрады. Ал ары қарай алгоритм құруды Пролог жүйесі автоматты түрде өзі жасайды. Ол оны өз бойындағы логикалық шығару механизмі көмегімен іске асырады. Бұл іс-әрекет былайша атқарылады: мәселенің мақсаты дерекқорға сұрату есебінде құрылады. Бұл сұратуда салалық мәселе аумағының сипаттамасы болады. Сұратудағы мәндерді дерекқордан іздеу үшін, Пролог жүйесі шығару механизмін іске қосады. Осылайша Прологта өтетін есептеулер есептің мақсатты тұжырымын дәлелдеуге арналған дедукция үдерісі болып табылады. Қазіргі кезде Пролог тілі де, оның негізіндегі теориялық тұжырымдамаларда көптеген салаларда кеңінен тарап, етек жайды. Ондай сала қатарына: *есептеу жүйелері архитектурасы*, программалау тілдері семантикасының *формальды сипаты*, дедуктивтік мүмкіндігі бар дерекқорлар сарапшы жүйелерді құру, лингвистика сияқты сала мәселелері жатады.

Сарапшы жүйелерді жобалауды автоматтандыру жүйелері. Мұндай жүйелер, жобаның өзін автоматтандыруға арналған құрал түріне жатады. Осындай құралдың мысалы ретінде TEIRESIAS жүйесін атауға болады. *TEIRESIAS жүйесі.* Программа құрамында *«Білім редакторы»* деп аталатын бөлігі болады. Ол үнемі ережелерді өңдеп, түзетіп отырады. Белгілі бір сала мәселесі бойынша үлкен білім қорын пайдалана отырып, жүйе осы білім қорын басқару үдерісін де қатар жүргізіп отырады. *TEIRESIAS* жүйесі білім қорындағы ережелерге сай белгілі бір шешім қабылдап, сол алған шешімді маманға сынауға ұсынады. Адам-маман жүйемен сұхбат нәтижесінде жаңа ережелерді тұжырымдап, ескі ережелерге өзгертулер енгізуі керек. Жүйенің осындай жұмыстарды жүргізіп отыруы «Ережелер үлгісі» деп аталатын құрылымдар көмегімен іске асады. Мұндай үлгілерді *мета-ережелер* деп атайды.

Қабықтар. Қазіргі уақытта ең тиімді құрал ретінде сарапшы жүйенің қабығы деп аталатын программалар пайдаланылады. Бұл

қабықтар белгілі бір салаға байланысты сарапшы жүйелерді құру ісін барынша жеңілдетеді. Дегенмен олардың да әлсіз жақтары бар, атап айтсақ, шешімді табу әдістері маман қолданған тәсілдерге және жүйеде қолданған білімді пайдалану үлгісі осы саладағы білімдер құрылымының үлгісіне дәл келмеуі мүмкін. СЖ-ні құрудың ең алғашқы кезеңдерінде сол кездегі бар программалау тілдері пайдаланылған. Одан кейінгі кезеңдерде білімдерді пайдалану тілдері деп аталатын құрылымдар қолданыла бастады. Бұл құрылымдар құрамында арнайы құралдардың екі түрі болады:

- білімдерді пайдалану үлгілері (ережелер немесе фрейм түрінде);
- интерпретатор – бұл құрал білімдерді пайдалану модульдерін белсенді түрге келтіруді басқарады.

Білімдерді пайдалану модульдерінің жинағы сарапшы жүйенің білім қорын береді де, ал интерпретатор құралы қисынды шығару машинасы деп аталады. Осындай құрылымдарды негізгі құрамдас бөлшек есебінде пайдалану, СЖ-нің қабығы деп аталатын құралдарды жасауға негіз болады. Қазіргі кезде классикалық деп атауға болатын сондай қабықтардың бірі – EMYCIN [32].

EMYCIN жүйесі. Бұл жүйенің атының өзі айтып тұрғандай, Empty MYCIN – бос MYCIN, 1981 жылы MYCIN жүйесі негізінде құрылған. Бұл қабық MYCIN жүйесінің барлық функционалды мүмкіндіктерін сақтап отырады. Соның нәтижесінде оны басқа сала мәселелерге қатысты білімдермен де толықтыруға болады. Қабықты одан әрі толықтырып, жұмыс істеу нәтижесінде бұрынғы түптұлғаға бірталай өзгерістер мен толықтырулар енгізілуі арқасында жобалар жұмысын автоматтандыруға мүмкіндігі бар *TEIRESIAS* жүйесі дүниеге келді. *EMYCIN* жүйесі көптеген сала есептерін шешуге мүмкіндік беретін консультациялық программаларды құруға арналған «қаңқа» қызметін атқара алатындықтан, EMYCIN қабығын дедуктивті түрде шешілетін есептерге қолдануға болады.

4.6. Сарапшы жүйе есептерінің жаттығулары

Сарапшы жүйе көмегімен шешілетін көптеген қолданбалы есептер бар. Сарапшы жүйе әдістерін қолданбас бұрын осындай мәселелерді шешуге қажет болатын келесі бағалау белгілерін есепке алып отыру керек.

1. Деректер мен білімдер сенімді және олар уақыт өткен сайын өзгермейді.
2. Мүмкін болатын шешімдер кеңістігі аса үлкен емес.

3. Есепті шешу кезінде белілі бір қалыпқа түскен тұжырымдар қолданылуы қажет. Қазіргі кезде әлі де болса ұқсастық пен абстракциялау әдістерімен шешіле бермейтін, білімдерге негізделген жүйелер бар.

4. Сала мәселесіне қатысты өз білімдерін жүйелі түрде түсіндіре алатын және осы білімін нақты есепті шешуге қолдана алатын ең аз дегенде бір сарапшы-маман адам болуы керек.

Осының алдындағы параграфтарда атап өткендей, сарапшы жүйе дегеніміз – адам-маманды белгілі бір салада ауыстыра алатын компьютерлік программа. Сала аумақтарының мысалы ретінде: медицинаны, экономиканы, ақпараттық жүйелерді, тамақ, жеңіл өнеркәсібін, т.б. салаларды атап өтуге болады. Осылардың бәріне қатысты орындалуға тиісті бір белгі бар: ол осындай сарапшы жүйелерді жобалау үшін адам-маманның болуы міндетті. Сарапшы жүйе құрамы жүйенің орындауға тиісті негізі функцияларын анықтап береді: жүйе пайдаланушымен адам тілінде сөйлесіп, адамша ұсыныс жасап, өз шешімін адам тілінде түсіндіріп, өз құрамындағы білімдерін адам баласы білімін қалай толықтырады, сондай тәсілмен толықтырып, өзіндегі ескі білімдерді де адам сияқты жөндейді. Сонымен қатар сарапшы жүйенің тағы бір ғажап қасиетіне өзіндегі жинақталған білімдерді ұзақ уақыт бойына сақтап, үнемі толықтырып отыруын жатқызумен бірге, ол өзінің білімін пайдаланып, қолданатын ұйымға біліктілік деңгейі жоғары мамандарды ауыстырып қана қоймай, осы ұйымға экономикалық тәуелсіздік әкеле де алады. Сарапшы жүйе программаларының басқа программалық өнімдерден тағы бір ерекшелігіне ол деректер, білімдер құрылымдарымен бірге арнайы шешімді шығару механизмін бөлек ұстап пайдалана білуін де жатқызамыз. Білімдер сарапшы жүйеде оларды есептеу машинасында оңай өңдеуге лайықты қалыпта орнатылады. Яғни сарапшы жүйеге есепті шешу алгоритмі емес, білімдерді өңдеу алгоритмі салынады. Сондықтан нақты мәселені шешу кезінде осындай білімдерді өңдеу алгоритмі тіпті бізге белгісіз шешім түрін де ұсына алады. Тіпті білімдерді өңдеу алгоритмі алдын ала белгісіз де болады, ол білім қорына салынған эвристикалық ережелер негізінде есеп шешімі кездейсоқ түрде алынады. Әрине, жоғарыда атап өткеніміздей, мәселені шешу кезінде сарапшы жүйе пайдаланушыға түсінікті тілде өз шешімін түсіндіріп отырумен бірге, маман адам алатын шешімнен кем түспейтін, тіпті кей кезде одан да артық болатын шешім түрін бере алады. Білімдерге негізделген жүйелерде нақты сала аумағына қатысты нақты білімдер жиыны ережелер (немесе эвристикалар) сарапшы жүйенің білім қорында сақталады. Шешілуге қажетті пробле-

ма жүйе алдына белгілі бір жағдайды сипаттайтын фактілер түрінде қойылады. Ал сарапшы жүйе өз білім корындағы білімдерге байланысты осы фактілерден қорытынды шығаруға тырысады. Сондықтан сарапшы жүйе сапасы оның білім қорындағы білімдердің саны мен сапасына тікелей байланысты болады. Сарапшы жүйе мынандай сатылық режимде қызмет атқарады: деректерді таңдау немесе талдау, бақылау нәтижелерін таңдау, алынатын нәтижені талдау, жаңа мағлұматтарды қорытып қабылдап алу, уақытша болжау ережелері көмегімен ғылыми болжамдарды ұсыну, содан соң деректердің келесі порциясын немесе талдау нәтижелерінің келесі тобын қабылдап алу. Бұл үдеріс соңғы қорытынды нәтиже толық болды немесе қанағаттанды деген хабар түскенше жүріп отырады.

Сарапшы жүйелерде кез келген уақыт сәттерінде білімнің үш түрлі типтері болуы мүмкін. Оларға мыналарды жатқызамыз: *құрылымдық, динамикалық, жұмысшы.* Құрылымдық білімдер – сала аумағына қатысты статикалық (тұрақты, өзгермейтін) білімдер. Бұл білмдер анықталғаннан кейін өзгермейді. Құрылымдық динамикалық білімдер – сала аумағына қатысты өзгеріп тұратын білімдер. Олар жаңа мағлұматтар түскен сайын жаңарып, өзгеріп тұрады. Жұмысшы білімдер – сала аумағындағы нақты бір есепті шешуге арналған немесе кеңес беруге бағытталған білімдер жиынтығы. Осы аталған білімдердің барлық түрлері сарапшы жүйелердің білім қорында сақталады. Мұндай жинақталған білім қорын құру, әрине, оңай жұмыс емес. Ол үшін адам-маманмен, оның біліміммен аса зор, ауқымды жұмыс атқарылуы керек. Содан соң осы маманмен бірігіп отырып, білім қорына кіретін білімдерді жүйелеу қажет, оны белгілі бір нұсқағыштар көмегімен ұйымдастырып, компьютер жадында сақтаудың құрылымдарын құру қажет. Осындай білім қорын құру жолдарының біріне білімдер үлгілерін құру жатады. Біз алдыңғы параграфтарда атап өткеніміздей, ол үлгілерді үш топқа бөліп қарастырамыз: продукциялық жүйе үлгісі, фреймдік үлгі, семантикалық желілер түріндегі үлгілер. Енді осы топтағы кейбір үлгілерге қатысты жаттығуларды қарастырайық.

4.6.1. Продукциялық сарапшы жүйелер құру

Жаттығу мақсаты: сарапшы жүйелерді жобалаудағы негізгі талаптар. Продукциялық үлгідегі сарапшы жүйелерді және оның бөліктерін жобалауды мысал арқылы үйрену.

Білімдерді ұсыну немесе пайдалануда кеңінен тараған тәсілдің біріне ондағы мағлұматтарды нақты *фактілер мен ережелер* түрінде

бейнелеу болып табылады. Әдетте, фактілер мынандай *үштік* түрінде сипатталады:

(АТРИБУТ НЫСАН МӘНІ).

Бұл факт мынаны білдіреді: берілген нысан өзінің атрибутымен (қасиетімен), өзіне қатысты оның белгілі бір мәнімен сипатталды. Мысалы, (ТЕМПЕРАТУРА ПАЦИЕНТ1 37.5) үштігі мына фактіні сипаттайды: «ПАЦИЕНТ1 деп аталған адамның температурасы, 37.5 деген мәнге тең». Кейбір қарапайым жағдайларда фактілер белгілі бір мәні бар атрибут арқылы емес, қарапайым тұжырыммен де сипаттала алады. Ол тұжырым *«шын»* немесе *«өтірік»* мәнін қабылдауы мүмкін. Мысалы: «Аспанда бұлттар көп». Мұндай жағдайларда фактіні бір қарапайым атаумен (мысалы, БҰЛТТАР) белгілеуге болады немесе осы тұжырымдағы мәтіннің өзін де алуға болады. Продукциялық үлгінің Білім қорындағы ережелердің түрі мынандай болады:

ЕГЕР А ОНДА S, мұндағы *A – шарт*; ***S*** *– әрекет.*

Бұл ережеде егер ***A*** шын болса, онда ***S*** әрекеті орындалады. Әдетте, S әрекеті, берілген шарт сияқты *тұжырым* бола алады, ондай тұжырымды, егер жүйеге ереженің А шарты шын (яғни жүйеге таныс) болған жағдайда жүйе *шығарып бере алады.*

Білім қорындағы ережелер маман адамның қызмет ету нәтижесінде алған тәжірибесінен туған *эвристикалық білімдерін (эвристикалар)*, білдіреді, яғни маманның әлі қалыпты түрге келе қоймаған ойлау тұжырымдарын білдіреді.

Осыған байланысты қарапайым өмірдегі бір мысалды қарастырайық. Мысалы, мынандай тұжырым бар делік:

ЕГЕР аспанды бұлт жауып тұрса,
ОНДА жақында жаңбыр жауады.

Бұл жерде ***A*** шарты есебінде бір факт (мысалдағыдай) немесе **«және»** логикалық операциясымен біріккен бірнеше A_1, ..., A_N, фактілер жүруі мүмкін:

A_1 **және** A_2 **және** ... **және** A_N.

Математикалық логикада мұндай өрнек *коньюнкция* деп аталады. Оның нәтижесі *«шын»* мәнін егер оның құрамындағы барлық бөліктері да *«шын»* мәнін қабылдаса ғана болады. Мысалы, жоғарыда аталған тұжырымның сәл күрделі түрін қарастырайық:

ЕГЕР
аспанды бұлт жауып тұрса **және** барометр тілі төмен түссе,
ОНДА
жақында жаңбыр жауады.

Ережелер құрамына кіретін әрекеттердің өзінде жаңа фактілер болуы мүмкін. Ондай ережелерді іске қосқанда, ондағы фактілер жүйеге белгілі болады. Ол фактілер жиынына қосылады, бұндай жиынды *жұмысшы жиыны* деп атайды. Мысалы, «аспанды бұлт жауып тұрса» және «барометр тілі төмен түссе» деген фактілер *жұмысшы жиынында* болса, онда аталған ереже іске қосылғанда, бұл жиынға «жақында жаңбыр жауады» фактісі де қосылады.

Білімді пайдаланудың продукциялық үлгісі. Мұндай үлгі түрінде білімдер келесі түрдегі ережелер жиындарынан тұрады: «ЕГЕР – ОНДА». Осындай үлгі түріне негізделген жүйелерді продукциялық жүйелер деп атайды. Бұндай жүйелерде шешімді шығару механизмі екі түрлі тәсілмен жұмыс істейді. Ол тәсілдерге: тура және кері шығару деп аталатын тәсілдер жатады. Кері шығару механизмі бар продукциялық жүйелерде ережелер көмегімен фактілер мен қорытындыларды бір бүтін есебінде жинақтайтын ЖӘНЕ/НЕМЕСЕ ағашы құрылады. Деректер қорындағы фактілер негізінде осы ағаш жапырақтарын бағалау нәтижесінде логикалық қорытынды шығарылады. Логикалық қорытындылар тура, кері және екі бағытты болып келеді. Тура шығару жолында іздеу нүктесі деректерден басталады да, бағалау үдерісі терістеуі бар түйіндерде тоқтауы мүмкін. Бұл жағдайда қажетті қорытынды есебінде ағаштың ең жоғарғы деңгейіне (түбірі) сәйкес келетін ғылыми болжам ұсынылады. Дегенмен мұндай шығару механизмі үшін деректердің өте көп мөлшерімен бірге ағаш жапырақтарын бағалау белгілері де қажет, ол кей кезде тіпті қажет болмаған жағдайда да іздеуді талап етеді. Кері шығару тәсілінің ерекшелігі қажетті қорытындыға қатынасы бар ағаш бөлігі ғана бағаланады. Бірақ бұл жолдың да өзіндік кемістігі бар: егер терістеулер мен тұжырымдар мүмкін болмаған жағдайда ағаш бұтақтарын одар ары қарай тудыру мүмкіндігі болмайды. Екі бағыттық шығару тәсілінде алдымен деректердің аз ғана көлемі бағаланады, содан соң ғылыми болжам таңдалады, содан кейін осы ғылыми болжамды қабылдау үшін қажетті деректердің басқа бөлігі сұратылады. Бұл шығару механизмінде күші қуатты және икемделгіш қасиеті бар жүйені алуға болады. Тура шығару тәсілін пайдаланатын жүйелер білімдерді пайдаланатын жүйелер ішіндегі ең алғашқы қолданылған жол болғандықтан, бұл тәсілді әдетте негізгі тәсіл деп қабылдайды. Бұндай алғашқы жүйелерде үш түрлі бөліктер болған: продукция жиынынан тұратын ережелер қоры, көптеген фактілерден тұратын деректер қоры және осы білімдерге негіздеп логикалық қорытынды шығаратын интерпретатор.

Ережелер қоры мен деректер қоры білімдер қорын құрады да, ал интерпретатор логикалық шығару механизмін құрады. Сарапшы жүйенің логикалық қорытынды шығару бөлігі – білім қоры мен жұмысшы жадысындағы мағлұматтармен жұмыс істеп, сондағы тұжырымдардың дұрыс-бұрыстығын тексеретін құрал. Ол қарапайым жүйелерде екі түрлі қызмет атқарады: жұмысшы зердесіндегі фактілер мен білім қорындағы ережелерді қарастырып, олардың «шын» не «өтірік» екенін көрсету үдерісін басқарады. Екінші атқаратын қызметі: ережелерді қарастырып, олардың іске қосылу тәртібін (приоритет) тексеріп, олардың қолдану мүмкіндігін қарастырады. Сонымен бірге жүйенің бұл бөлігі жүйе орындайтын барлық үдерістерді: кеңес беру, анықтама қызметі, пайдаланушыға арналған сеанс ақпаратын сақтау, онымен келіссөз жүргізу сияқты көптеген үдерістерді басқару қызметіне қатысады. Шығару әдетте «түсіну – орындау» кезеңі түрінде орындалады және әр кезеңде таңдап алынған ережелер деректер қорын түзетіп отырады. Соның нәтижесінде деректер қорының мазмұны бастапқы күйден мақсатты күйге қарай жылжиды, яғни мақсатты жүйе деректер қорында қалыптасады. Басқаша айтқанда, продукциялар жүйесі үшін қарапайым таңдау кезеңі мен ережелерді орындау кезеңі жұмыс істейді. Дегенмен жұмыс істеу барысында жүйедегі ережелер қорындағы ережелермен үнемі салыстыру болып отырғандықтан, егер ережелер саны көбейіп отырса, шығару жылдамдығы азаяды. Енді осы продукциялық үлгіні қолданатын жүйелердің күшті және әлсіз жақтарын тағы бір пысықтап өтейік.

Күшті жағы: жеке ережелерді құру және түсіну өте қарапайым, әрі ыңғайлы; осындай ережелерді толықтыру, жөндеу, алып тастау, қосу мүмкіндіктері де қарапайым; логикалық қорытынды шығару механизмі де түсінікті, әрі қарапайым.

Әлсіз жағы: ережелердің өзара байланысын орнату қиын; білім бейнесінің толық мүмкіндігін бағалау қиындығы; өңдеу деңгейінің өте төмендігі; адам білімінің құрылымынан өзгешелігі; логикалық қорытынды шығарудың икемсіздігі.

Егер зерттеу нысаны есебінде қарастырылатын мәселенің көлемі аса үлкен болмаса, онда продукциялық жүйелердің күшті жақтары белең алады. Егер білім мөлшері көп, шығаруда жоғарғы жылдамдықты қажет ететін және икемді шығару механизмін қажет ететін күрделі есептерді шығара бастасақ, онда жұмыс істейтін деректер қорында құрылымдық өзгерістер қажеттігі туып, продукциялық жүйелер өз әлсіздігін көрсете бастайды.

Сонымен, продукциялық үлгі түрі:

(i); P; Q: Ai => Bj ; N;

Мұндағы ***i*** – ереже нөмірі, ***P*** – ереженің артықшылығы, ***Q*** – ережелерді қолдану аумағы $A_i \Rightarrow B_j$ – продукция ядросы, ***i, j*** – тұжырымның => ***(секвенция)*** белгісінің қай жағынан алатынын көрсететін параметрлер. Әдетте, ***i*** – Дерекқор, Білім Қоры, Сұхбат, О (түсіндіру блогы) сияқты бөліктерден, ал ***j*** – оның алынатын жақтары да сондай, содан басқа оған білімдер толықтыру блогы енуі мүмкін, N – продукцияларға арналған түсіндірмелер. Мысалы, мынандай болып келуі мүмкін: $A_{БД} \Rightarrow B_{БД}$ немесе $A_{БД} \Rightarrow B_{БЗ}$ немесе $A_{Д} \Rightarrow B_{БЗ}$.

Бақылау мысалы 1. «Консалтингтік қызмет көрсету» деп аталатын сарапшы жүйені құру кезінде продукциялық үлгіні пайдаланамыз. Жүйенің білім қорындағы бір ережені қарастырайық. Оның сөйлеммен берілген сипаты:

ЕГЕР процессор аты = «Celeron» ЖӘНЕ жады көлемі = 256 болса, ОНДА осы параметрлері бар барлық компьютерлердің деректерін бер.

Ереженің SQL сұрату тіліндегі сипаты келесі түрде өрнектеледі:

*SELECT * FROM basic WHERE proc like 'Celeron%' and memory='256' ORDER BY art*

Ереженің продукция ядросының түрі мынандай болады:

$$A1_{БД} \text{ ЖӘНЕ } A2_{БД} \Rightarrow B_{БЗ}$$

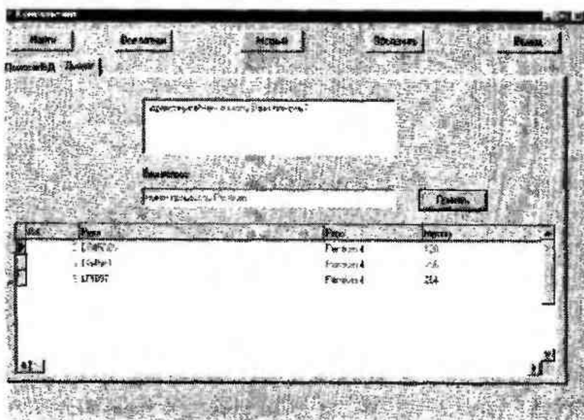
Бұл жердегі $A1_{БД}$ = *процессор аты = «Celeron» фактісі де,* $A2_{БД}$ = = *жады көлемі = 256* фактісі болады, ал $B_{БЗ}$ = *осы параметрлері бар барлық компьютерлердің деректерін бер* фактісі секвенция белгісінің сол жағында орналасады. Яғни секвенция (⇒) белгісінің сол жағындағы фактілер жұмысшы жиынынан, ал оң жағындағы білім қорынан алынады. Мұнда жұмысшы жиыны есебінде кәдімгі реляциялық дерекқорды алуға болады. Ал *SQL* сұратуын орындағанда фактілер былайша қалыптасады: $A1_{БД}$ = *proc like 'Celeron%'* және *(and)* $A2_{БД}$ = *memory* = = *'256'* болып, ал шығару бөлігінде $B_{БЗ}$ = ** FROM basic* түрінде болады.

Енді сарапшы жүйе бөліктерінің бірі – *Сұхбатты* қалай құратынымызды қарастырайық. Сонымен бірге осы бөліктің мақсатын, оның басқа бөліктермен өзара әсерін шола кетеміз. Әдетте, нақты салаға лайықталған сарапшы жүйелерде Сұхбат табиғи тіл ортасында құрылып, оны басқарушы программа модулін сұхбат процессоры деп атайды. Сарапшы жүйелерде Сұхбат блогын ұйымдастыруға арналған екі түрлі форма бар: біріншісі – шектелген табиғи тілдегі жиынды пайдалану, екіншісі – бірнеше әрекетті алдын ала таңдау нәтижесінде сұхбат жүргізу. Бірінші формада *сөздік-меню* деп аталатын құрылымды пайдалану. Онда маман-адамның сөйлем қорынан, яғни лексиконынан тұратын сөйлемдерден *негізгі* сөздерді таңдап алып, әрекет жасау. Екіншісінде белгілі бір іс-әрекетті беретін сөздерді алдын ала дайындап, пайдаланушыға ұсыну. Сұхбат блогында тағы бір ескере кететін жағдай – ол сарапшы жүйені пайдаланатын адамдар санаты. Жалпы жағдайда сарапшы жүйеде пайдаланушының екі санаты бар. Олардың жүйеден талап ететін мақсаттары әртүрлі болғандықтан, пайдаланушылар мынандай екі түрге бөлінеді:

1) әдеттегі пайдаланушы, оған сарапшы жүйеден *кеңес* қажет, сондықтан ол жүйемен адамша сөйлесіп, өзіне қажетті мағлұматты алады.
2) білім инженериясына жататын топ. Оларға саладағы маман адам және білім инженерін қосуға болады. Бұл топтың қажетін сарапшы жүйенің білімді толықтыру бөлігі деп аталатын бөлігі орындайды. Ол компонента сұхбат жүргізу нәтижесінде білімдерді толықтырып, жөндейді, өзгертеді.

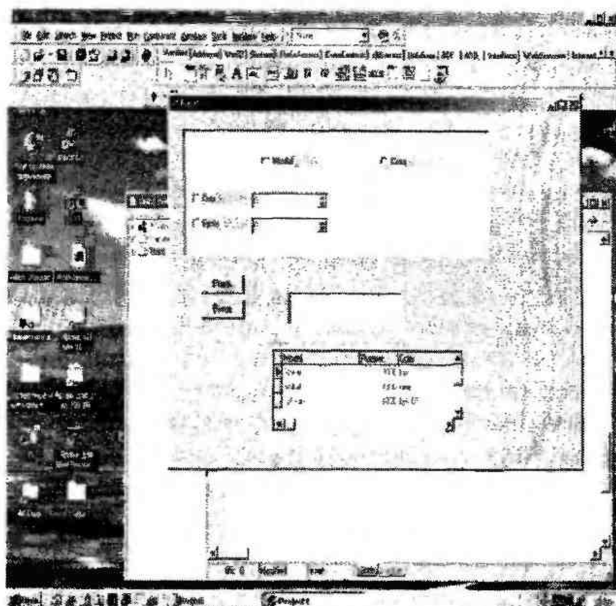
Жоғарыда келтірілген «Консалтингтік қызмет көрсету» деп аталатын сарапшы жүйені құру кезінде оның Сұхбат бөлігін жасау үшін, негізгі сөздер механизмін пайдаландық. Сонымен, пайдаланушының жүйемен сұхбаты келесі түрде болуы мүмкін: (4.6.1-суретті қара).

Мысалда көрсетілгендей (Мысал программалық Дельфий ортасында орындалған), пайдаланушы жүйеге берген жауабында («нужен процессор») негізгі сөз бар. Ол – «процессор» сөзі. Осы сөз басқаруды білім қорындағы ***SQL*** сұратуын іске қосуда пайдаланылады. Егер пайдаланушы енгізген сөйлемде негізгі сөз болмаса, онда жүйе қосымша анықтаманы қосу арқылы пайдаланушыны бәрібір жетектеп, негізгі сөздерге алып келеді.



4.6.1-сурет. Сұхбат блогының принскрині[1]

Сұхбат жүргізіліп болған соң, экранға білім қорындағы ережелер арқылы таңдалған сұрату нәтижесі шығады. (4.6.2-суретті қара).



4.6.2-сурет. Кеңес берудің принскрині

[1] Принскрин – ақпараттың компьютер терезесінен көшірмесі. Көбінде суреттерді көшіруге пайдаланылады.

Бақылау мысалы 2. Тағы бір сала мәселесі бар делік. Ол адамдарды таныстыру, кездестіру, достастыру, үйлендіру саласында жұмыс істейтін таныстыру агенттігіне қатысты мәселе. Сарапшы жүйенің аты – «Электрондық құдағи». Сарапшы жүйенің Сұхбат бөлігін мынандай кілттік сөздер: «жұп», «қыз» (немесе «ер адам», «әйел адам»), «ақпарат» арқылы басқарайық. Кілттік сөздер механизмін ұйымдастыру былайша орындалған. Сұхбат жүргізгенде жүйе жауап беретіндей кілттік сөздерді ұйымдастыру құрылымы болуы қажет. Сондай құрылым есебінде 4.6.1-кестесінде келтірілген кілттік сөздердің топтасуын қарастыруға болады. Бұл топтасу осы мысалда кесте көмегімен алынған. Кестенің екі бағанында жүйеге қойылатын сұрақтар мен оларға әсер ететін кілттік сөздерді бір жазба түрінде (қатар) орналастырып, сұхбатты осы кестеге байланысты жүргізеді. Жүйе реакциясы бағанында жүйенің қоятын мүмкін болатын сұрақтары мен оған мүмкін болатын жауаптар екінші кілттік сөздер бағанында орналасады. Кесте аты – Questions.db. Бұл кесте кілттік сөздер мен сұхбат сөйлемдерінің байланыстарын реляциялық түрде деп түсіндіреді.

4.6.1-кесте

Questions.db

Жүйе реакциясы	*Негізгі сөз*
Қандай?	ақпарат
Кез келген бе, әлде ерекше ме?	Ер адам
Онда ерекшеліктерін ата (бойы, жасы, көзінің түсі, шашының түсі)	ерекшелігі
Сұратуға тағы қосасыз ба?	бойы

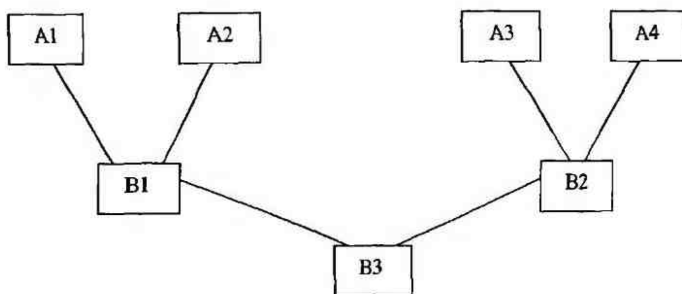
Қойылған сұрақтардан соң оларға берілген жауаптарды талдау басталады. Осы талдау нәтижесінде басты сөздер бағанындағы сөздермен пайдаланушы жауабы салыстырылады. Егер сөз дөп келсе, сұхбат не одан ары қарай жүреді немесе басқару Білім қорының сұратуына түседі. Осылайша сұхбат арқылы басқарылған программа модулі іске қосылады.

Сарапшы жүйенің тағы бір маңызды бөлігіне білімді толықтыру бөлігі жатады. Бұл блок адам баласының өзіндегі білімді қалай толықтыратын принципі бойынша жұмыс істеуі қажет. Яғни компьютерлік программа білімді ала білу өнерін адам сияқты меңгеруі қажет. Оның тағы бір қасиеті – ол сарапшы жүйедегі ескі білімдерді

де жөндеу дағдысын үйренуі керек. Қарапайым жүйелерде жаңа ережелерді білім қорына қосу ғана орындалса, одан күрделі сарапшы жүйелерде білім қорындағы бар ережелер жөндеуге түсу мүмкіндігіне ие болады. Сонымен бірге күрделі жүйелерде жөнделген, жаңадан енгізілген ережелер бұрынғы ережелермен қайшылыққа түспеуін қадағалайтын тексеру модульдері де болады. Жүйенің бұл блогын басқару үшін, оның жүйенің басқа компоненталармен болатын байланыстарын білу қажет. Мысалы, бұл бөлік міндетті түрде Сұхбат бөлігімен байланыста болады. Өйткені оның интерфейсі табиғи тілде өтетіндіктен, негізгі сөз қорының сөйлемдері осы блокта сақталуы мүмкін. Сондықтан осы блоктағы қабылданған механизмдерді түсіндіру бөлігінде де пайдалануға болады. Жүйенің білім қорымен түсіндіру блогы арасындағы байланыс түрі тіпті тығыз деп айтуға болады. Ол үшін түсіндіру модульдері әдетте екі түрлі сұрақ төңірегінде құрылады. Ол сұрақтар – «Осы айтқан кеңесіңіз ***қалай*** пайда болды?» және «Сарапшы жүйе ***неге*** осындай шешімге келді?». Осы екі сұраққа жауап беру нәтижелерінің механизмдерін сарапшы жүйенің түсіндіру блогы басқарады. Жүйе ***«Неге?»*** деген сұраққа жауап беруі үшін, ол жауап жолын соңғы қолданылған ережеден бастайды, яғни шешім қабылдау үдерісінің ең соңғы қадамынан бастайды. Мұндағы ***«Неге?»*** сұрағы жүйенің тұжырымға келу жолын ереже шартынан бастап, ереже қорытындысына қарай бағытталған жолмен алып жүреді. Дегенмен шығару үдерісін басқа жағынан да зерттеу үшін, оның ережелердегі кері тұжырымға келуін де қарастыру қажеттігі бар. Яғни тұжырымды талдауды оның қорытынды бөлігінен бастап, шарт бөлігіне қарай жүру жолын қарастыру. Осындай талдауды іске асыратын сұраққа ***«Қалай?»*** сұрағы жатады. Осы ***«Қалай?»*** сұрағына жауап беру нәтижесінде сарапшы жүйе пайдаланушыға қолданған білім қорының ережелерін көрсете отырып, қадамнан соң қадам арқылы шешімді алудың бүкіл барысын көрсетіп береді. Шығаруды толық бейнелеп көрсету үшін бір мысал қарастырайық. Шығару желісі бірнеше П1, П2, П3 деген атаулары бар ережелерден тұрады дейік. Оның продукциялық үлгідегі көріністері келесідей болсын:

- П1: ЕГЕР А1 ЖӘНЕ А2 шын болса, ОНДА В1 – шын;
- П2: ЕГЕР А3 ЖӘНЕ А4 шын болса, то В2 – шын;
- П3: ЕГЕР В1 ЖӘНЕ В2 шын болса, то В3 – шын.

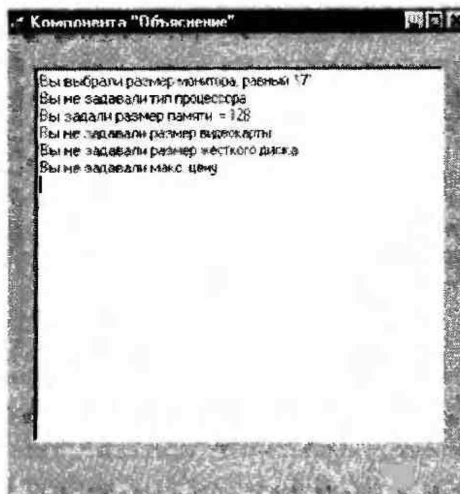
Ал осы ережелер бір-бірімен желі арқылы байланысқан (4.6.3-суретті қара).



4.6.3-сурет. Түсіндіру блогының сызбанұсқасы

Әр ережеге оның шарт бөлігі мен қорытынды бөлігіне сөйлемдерден тұратын түсіндіру мәтіндерін байлап қоямыз. Яғни бұл мәтіндерді шаблондар түрінде ұйымдастыруға болады. Кез келген уақытта сұхбат арқылы сарапшы жүйенің кеңес бергенін көрсететін шығару жолын экранда көрсетуге болады. Сұхбат бөлігі түсіндіру бөлігіндегі шаблон-өрістердегі мәтіндерді шығарып бере алады.

Мысалы, жоғарыда келтірілген «Консалтингтік кызмет көрсету» деп аталатын сарапшы жүйені құру кезінде оның түсіндіру бөлігін жасау үшін «Неге?» деген сұраққа жауап іздейтін механизмді пайдаландық. Сонымен, пайдаланушының жүйеге түсіндіруі келесі түрде болуы мүмкін (4.6.4-суретін қара).



4.6.4-сурет. Түсіндіру блогының принскрині

Сарапшы жүйенің бөліктерін продукциялық үлгі негізінде құруға арналған жаттығуларды меңгеруге қажетті терминдерге мыналарды жатқызамыз: ***сарапшы жүйелер, сарапшы жүйе бөліктері, сарапшы жүйе дерекқоры, білім қоры, білімді пайдалану үлгілері, логикалық үлгі, продукциялық үлгі, сұхбат компонентасының механизмдері, түсіндіру бөліктерінің механизмдері, шығару стратегиясының механизмдері.***

Сарапшы жүйені продукциялық үлгімен жобалауға арналған тапсырмалар:

1. «Электрондық құдағи» продукциялық үлгісін жобалау.
2. Осы сарапшы жүйеге арналған Дерекқор, Білім қоры блоктарын құру.
3. Алдыңғы сарапшы жүйеге арналған «Сұхбат» блогын құру.
4. Алдыңғы сарапшы жүйеге арналған «Түсіндіру» блогын құру.
5. Алдыңғы сарапшы жүйеге арналған «Толықтыру» блогын құру.
6. Өзіңіз ойлап тапқан салаға қатысты продукциялық үлгісін жобалау.
7. Осы салаға қатысты сарапшы жүйенің барлық бөліктерін құру.

4.6.2. Фреймдік сарапшы жүйені құру

Фрейм (*ағылш. Frame* – рамка, каркас – шетпер, қалып) ұғымы – адам зейіні қабылдайтын түсініктердің жадыда қалыптасатын абстракты бейнесі. Әр фреймнің өзіндік атауы және өз слоттармен олардың мәндерін қамтитын тізімдері болады. *Слот* (*ағылш. slot* – щель, прорезь – тесік, кесік) терминал болуы да мүмкін, (иерархия жапырағы) немесе төменгі деңгейдегі фрейм болуы да мүмкін. Фрейм мәндері ретінде деректердің кез келген түрлері мен басқа фрейм атаулары да бола алады. Осылайша фреймдер өзара желілер құра да алады. Сонымен бірге фреймдер арасында АКО (a kind of), типіндегі байланыс түрлері бар. Бұл байланыс түрі фреймнің одан да жоғарғы деңгейдегі байланыс түрі бар екенін көрсетеді. Бұл жоғарғы деңгейдегі фреймде слоттар мәні мен тізімдер орналасады. Бұл үдерісті мұрагерлік деп атайды. Егер мұрагерлік бірнеше түптұлғадан таралса, онда жиындық мұрагерлік түрі орын алады. Кез келген фреймді былайша құруға болады:

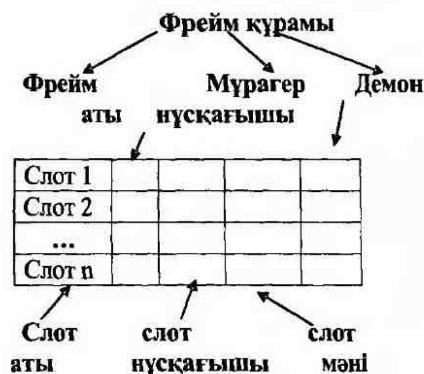
(ФРЕЙМ АТЫ:
(1-ші слот аты: 1-ші слот мәні),
(2-ші слот аты: 2-ші слот мәні),

....

(N-ші слот аты: N-ші слот мәні)).

Негізгі ұғымдар мен терминдер. Фреймдік үлгі адам жадысы мен оның есі-санасының жүйеленген психологиялық үлгісі ретінде ұсынылған. Әрбір фреймде көптеген кез келген мөлшердегі слоттар болады, оның кейбіреулерін арнайы функцияларды орындау үшін жүйенің өзі тағайындаса, қалғанын пайдаланушы анықтай алады. Фреймдер, әдетте, стандартты жағдайларды бейнелеуге арналған білімдер фрагментін сипаттайды. Осы ситуациялардың белгілі бір құрылымдық бөліктерін слоттар деп атаған. Слоттар басқа фреймді де бейнелеуі мүмкін, мұндай жағдайда екі фрейм арасында белгілі бір байланыс пайда болады. Әрбір фрейм құрылым есебінде сала мәселесіне қатысты білімдерді сақтайды. Оны фрейм-түптұлға деп атайды. Ал фрейм слоттары нақты мәндермен толықтырылғанда, олар белгілі бір уақиғаны немесе үдерісті суреттейтін нақты фреймге айналады. Фреймнің басқа құрылымдардан ерекшелігі – ол оған мүмкін болатын және пайда болуы мүмкін жағдайларды үлгілеуге болатындығы. Ол слоттарға үнсіз келісім бойынша стандартты жағдайларды тағайындап коюға байланысты болады. Іздеу үдерісі кезінде бұл мәндер одан да нақтырақ мағыналарымен толықтырылуы мүмкін. Кейбір айнымалыларды жүйе бөлектеп қарайды. Өйткені оның мәндері жөніндегі мағлұматты жүйе пайдаланушыдан сұрау арқылы алады. Кейбір айнымалылар кіріктірілген процедуралар көмегімен анықталады. Ол процедураларды кейде ішкі процедуралар деп атайды. Айнымалыларға нақты мәндер беру арқасында басқа да процедуралар шақырылып тұрады. Айнымалылар типтерін бұлайша бейнелеу мағлұмдамалық және процедуралық білімдерді араластырып пайдалануға мүмкіндік береді. Көптеген әртүрлі сала аумақтарының есептері үшін фреймдік үлгі білімдерді қалыптастырудың негізгі жолы болып тұр. Жалпы фреймнің өзі белгілі бір жүйені береді. Яғни фреймдік жүйе – иерархиялык құрылым. Оның түйіндерінде басқа фреймдер орналасуы мүмкін. (4-бөлімдегі иерархиялық фреймді қара). Енді осы фреймге кіретін деректердің құрылымдық сипатын қарастырайық. (4.6.5-суретті қара). Осы суретте бейнеленген әр элементтің қысқаша сипатын анықтайық.

1) *Фрейм аты.* Бұл фреймге берілетін идентификатор әр фреймге беріледі, ол – аталған фрейм жүйесіндегі осы фрейм үшін ерекше, басқаға ұқсамайтын бірегей атау. Әр фреймде көптеген кез келген мөлшердегі слоттар болады, оның кейбіреулерін арнайы функцияларды орындау үшін жүйенің өзі тағайындаса, қалғанын пайдаланушы анықтай алады. Осындай слоттардың біріне IS-A жатады, ол осы фреймнің ата-фреймін көрсетеді, ал оның слоты өз бойында балалық фреймдер нұсқағыштарын ұстайды. Слоттағы бұл нұсқағыштар фреймдер нұсқағыштарының тізімдері болып табылады, мысалы, пайдаланушы атын енгізу слоты, фреймді анықтау мерзімінің слоты, өзгерту мерзімінің слоты, түсіндірме мәтінінің слоты сияқты слоттарды атауға болады. Әрбір слот өз кезегінде тағы да деректердің белгілі бір құрылымымен сипатталады.



4.6.5-сурет. Фрейм деректерінің құрылымы

2) *Слот аты.* Бұл слотқа берілетін идентификатор, слоттың өзі де фрейм сияқты ерекше, басқаға ұқсамайтын атауға ие болуы қажет. Әдетте, слот атауында ешқандай мағына болмайды, ол осы слоттың идентификаторы болады. Дегенмен кейбір жағдайларда оның арнайы мағынасы болуы да мүмкін. Мысалы, IS-A (IS-A қатынасы), DDESENDANTS (тура еншілес фреймнің нұсқағышы), FINEDBY (фреймді анықтайтын пайдаланушы), DEFINEDON (фреймді анықтау мерзімі), MODIFIEDON (фреймді өзгерту мерзімі), COMMENT (комментарий) және тағы басқа осы сияқты атаулар құрылымдық нысандарды сипаттай алады. Осындай құрылымдық нысандарға HASPART, RELATIONS деп аталатын нысандарды жатқызуға болады. Әдетте, бұл слоттар жүйелік деп аталып, оларды білім қорын жөндеп

өзгерткенде және шығаруды басқаруда пайдаланады. Слоттың мәнін алу тәсілі слотқа қандай нақты мән берілетінін анықтайды. Осындай тәсілдердің бірнешеуі бар және ол деректердің қасиетіне байланысты болады. *Слоттар мәндерін алу тәсілдері.* ***1) үнсіз келісім бойынша түптұлғадан алу*** *(атадан).* Слотқа фрейм-түптұлғасында үнсіз келісім бойынша анықталған мән беріледі, ол мән стандартты болуы да мүмкін. ***2) мұрагерлік арқылы.*** Бірінші тәсілден ерекшелігі мән ағымдағы АКО байланысымен қосақталған аталық фреймнің арнайы слотында беріледі. ***3) формула бойынша.*** Слотқа белгілі бір формула тағайындалып, сол бойынша есептелген нәтиже слот мәні болады. ***4) қосақталған (присоединенная) процедура бойынша.*** Слотқа белгілі бір процедура тағайындалып, сол бойынша слот мәні алгоритм арқылы алынады. ***5) деректердің сыртқы көздерінен.*** Зерделік жүйелердегі үлгілерді пайдаланғанда, слоттар мәні болып табылатын деректер әртүрлі көздерден, мысалы, дерекқордан, датчик жүйелерінен, пайдаланушыдан түсуі мүмкін. Фреймдер теориясында слоттарға әртүрлі арнайы процедуралардың қосақталуы мүмкін болатын жағдай. Ол үшін демон деп аталатын құрылымдар пайдаланылады. ***6) демон*** деп белгілі бір шарт орындалғанда соған сәйкес автоматты түрде іске қосылатын процедураны айтамыз. Демондардың бірнеше түрі бар. Осындай қосақталған процедуралар механизмдері жағынан реляциялық дерекқорлардағы триггерлер процедураларына ұқсас болып келеді.

3) *Мұрагерлік нұсқағыштар.* Бұл нұсқағыштар иерархиялық типтегі фреймдік жүйелерде болады. Бұл жүйелерде «абстракты – нақты» байланыстары сипатталады, олар төменгі деңгейдегі фреймдердегі слоттар мәні осы аттас жоғарғы деңгейдегі фреймнің слоттарынын атрибуттары қандай мағлұматтарды қамтитыны жайындағы ақпаратты өз бойында сақтайды. Типтік мұрагерлік нұсқағыштарға: Unique (U: бірегей), Same (S: осындай), Range (R: шекара орнатады), Override (O: есепке алмайды) тағы басқалары жатады. Мұндағы U нұсқағышы әрбір фрейм әртүрлі мәнді слоттарды өз бойында сақтай алатынын білдіреді: S – барлық слоттардағы мәндер бірдей, R – төменгі деңгейдегі фрейм слотының мәні жоғарғы деңгейдегі фреймдер слоттарында көрсетілген шектерден аспауы қажет, O егер төменгі деңгейдегі фрейм слотының мәні анықталмаса, онда оның мәні есебінде жоғарғы деңгей фрейм слотының мәні қабылданады. O сонымен бірге U және S нұсқағыштарының қызметтерін бір мезетте орындай алады. Көптеген жүйелерде мұрагерлік нұсқағыштардың бірнеше нұсқасын пайдалану мүмкіндігі бола тұрса да, кей жағдайда

мұндай нұсқағыштардың тек біреуі ғана қолданылады. Бұл жағдайда үнсіз келісім бойынша О нұсқағышы қолданылады.

4) *Деректер типтерін көрсету.* Слот сандық мәні бар екені көрсетіледі немесе слот басқа фреймнің нұсқағышы (яғни фрейм аты көрсетіледі) есебінде жүре алады. Деректер типтеріне мыналар жатады: FRAME (нұсқағыш), INTEGER (бүтін), REAL (нақты шама), BOOL (бульдық шама), LISP (қосақталған процедура), TEXT (мәтін), LIST (тізім), TABLE (кесте), EXPRESSION (өрнек) және тағы басқалары.

5) *Слот мәні.* Бұл слот мәнін енгізу пунктісі. Слот мәні осы слоттағы көрсетілген деректер типтерімен сәйкес келуімен қатар, мұрагерлік шартының да орындалуы қажет.

6) *Демон.* Бұл жерде мынандай типтегі демондар анықталады: IF-NEEDED, IF-ADDED, IF-REMOVED және тағы басқа осы сияқтылар. Демон деп белгілі бір шарт орындалғанда іске қосылатын процедураны айтамыз. Демондар белгілі бір слотқа қатынағанда іске қосылуы мүмкін. Мысалы, IF-NEEDED демоны егер белгілі бір уақыт сәтінде слотқа қатынау кезінде оның мәні анықталмаған жағдайда іске қосылады. Сол сияқты IF-ADDED демоны слотқа белгілі бір мән берілгенде іске қосылса, ал IF-REMOVED демоны слот мәнін өшіргенде іске қосылады. Сонымен бірге демон қосылған процедураның басқа бір түрі болып табылады. Осы жоғарыда аталған демондардың барлығы кеңінен тараған демондар түріне жатады. Қысқаша демондар сипатын тағы бір айта кетелік. IF-REMOVED слоттан ақпарат өшірілгенде, IF-ADDED слотқа жаңа мән қосылғанда, IF-NEEDED талап бойынша орындалады. Соңғы демон бос слоттан ақпарат сұратылғанда іске қосылады. Ал IF-DEFAULT демоны үнсіз келісім бойынша орындалады.

7) *Қосақталған процедура.* Слот мәні есебінде процедуралық типтегі программаны да пайдалануға болады. Бұл процедура, мысалы, Лисп тілінде қызметтік процедура (servant) деп аталады, ал Смолток тілінде белгілі бір әдісті білдіреді. Жалпы жағдайда қосақталған процедура басқа фреймнен келіп түскен хабардан кейін іске қосылады. Егер біз білімдерді пайдаланудың фреймдік үлгілерінде процедуралық және мағлұмдамалық білімдер қосылады десек, онда жоғарыда аталған демондар мен қосақталған процедураларды процедуралық білімдер деп қарастырамыз. Тағы бір айта кететін жайт – фреймдік үлгілерде шығаруды басқарудың арнайы механизмдері жоқ, сондықтан пайдаланушы бұл олқылықты осы қосақталған процедура арқылы толтыра алады. Сонымен қоса бұл қосақталу процедурасының жан-

жақты қасиеті бар, сондықтан оның фреймдік үлгідегі білімдерді пайдаланудағы иерархиялық және желілік көрсетімдерінен де басқа түрлерін сипаттауға қуаты жетеді. Яғни қосақталған процедуралар көмегімен кез келген программаны жазуға болады. Бірақ ол пайдаланушы үшін қосымша жүктеме болады. Жасанды зерде бағытындағы күрделі қолданба есептерін шешетін мамандар үшін таптырмайтын құралдардың бір түрі деуге болады.

Фреймдер типтері. Фреймдер мынандай типтерге бөлінеді:

- **фрейм-экземпляр** – сала мәселесінің ағымдағы күйін сипаттайтын фреймнің нақты іске қосылған түрі;
- **фрейм-образец** – сала мәселесіндегі мүмкін болатын жағдайлар мен нысандарын сипаттайтын шаблон;
- **фрейм-класс** – фрейм-образецтердің жиынын сипаттауға арналған жоғарғы деңгейдегі фрейм.

Әр фреймдік үлгідегі фреймдер мен слоттардың саны да, құрамы да әртүрлі болуы мүмкін. Дегенмен егер бір фреймдік жүйе құратын болсақ, олардың біркелкі болуы үлгіні құруды жеңілдетеді. Ал «Әлем әркелкілігі» тұжырымдамасына сәйкес әртүрлі нысандарды біріктіруде белгілі бір келісім болуы қажет. Мысалы, ат біздің нақты әлемімізде қанаты жоқ жануарлар түріне кірсе, ал аңыз әлемінде оның қанаты бар, мысалы, «Ертөстік» ертегісіндегі Шалқұйрық жылқысы. Егер біз жылқының осы атауларын фрейм түрінде сипаттау қажеттігі болғанда екі түрлі әлемді міндетті түрде қарастырар едік. Жалпы жағдайда фреймдік үлгінің мағлұмдамалық және процедуралық білімдерінің барлық қасиеттерін сипаттауға күші жетеді. Слоттардың фреймдердегі бір-біріне ену деңгейлері (тереңдік деңгейлері) сала аумағының қасиеттері мен үлгіні іске асыруға болатын тіл мүмкіндігіне байланысты болады. ***Фрейм-прототиптер*** (шаблондар, үлгілер) сала аумағындағы абстракты ұғымдардың білімін бейнелейді, олар нақты объектілердің кластары болып табылады. Мысалы: «адам», «автокөлік» деген сияқты. ***Фрейм-экземплярлар*** сала аумағындағы нақты ұғымдардың білімін бейнелейді, яғни ол саладағы фактілерді сипаттайды. Мысалы, «Талқанбаев Қайрат», оның «Мерседес» маркалы автокөлігі бар. Атқаратын қызметіне байланысты фреймдер мына түрлерге бөлінеді: ***Фреймдер-нысандар*** бұлар сала аумағына қатысты абстракты және нақты заттар мен ұғымдарды мағлұмдамалық түрде сипаттайды, яғни нысанды немесе ұғымды сипаттайтын қасиеттер жиынын білдіреді. Мысалы: қарыз, вексель, адам, дәріс деген ұғымдар. ***Фреймдер-операциялар*** сала аумағына қатысты әртүрлі түрлендіру

үдерістерін бейнелейді, яғни үдерісті сипаттайтын қасиеттер жиынын білдіреді. Мысалы, қарызды алу үдерісі, вексельді толтыру, адамды үйрету, дәрісті оқу деген сияқты. ***Фреймдер-Ситуациялар*** сала аумағына қатысты типтік ситуацияларды прагматикалық түрде бейнелейді. Бұндай ситуацияға ***фреймдер нысандар*** және ***фреймдер рөлдер*** (олар жағдайды білдіретін сипаттарды өз бойында ұстайды) түседі. Мысалы: апат, қорқыныш, құрылғының жұмыс режимі деген сияқты. ***Фреймдер-Сценарийлер*** олар технологиялық сипатта болады да, белгілі бір іс-әрекетті, ұғымды, уақиғаны бейнелейтін типтік жағдайды сипаттайды, белгілі бір әрекеттердің динамикасын, яғни өзгеріп тұруын да сипаттайды, яғни жүйенің белгілі бір сценарий бойынша жылжуын бейнелеуі мүмкін. Мысалы: туған күнді тойлау, емтиханды тапсыру деген сияқты. ***Фреймдер-рөлдер*** ол функционалды болып келеді де, фрейм-нысанның белгілі бір жағдайда орындайтын типті рөлін бейнелейді. Мсыалы: менеджер, кассир, клиент, студент, оқытушы.

Бақылау мысалы 1. *Есептің қойылымы. Білімдерді пайдаланудың фреймдік үлгісі негізінде құрылған іздеу жүйесі.* **Мақсаты.** Білімдерді фреймдер арқылы пайдалануды және оны басқару тәсілдерін зерттеу. Білімдерді көрсетудің келесі талаптарға сәйкес тілін құру:

- фрейм бірліктерінде білімдерді көрсету;
- абстракция дәрежесіне негізделген фреймдердің иерархиялық құрылымын енгізу;
- мағлұмдамалық және процедуралык білімдердің араласкан көрсетімін беру мүмкіндігі.

Фреймдер деректерінің базалық құрылымында мыналар болуы қажет: Фрейм атауы, Слот атауы, Мұрагерлік нұсқағыштары, Деректер типтерін көрстекіштері, Слот мәні, Демон, Қосақталған процедуралар. *Шығаруды басқарудың үш тәсілін пайдалану қажет:* қосақталған процедуралар-демондар көмегімен; қызметтік процедуралар көмегімен; мұрагерлік механизмдер көмегімен.

Мысалды шешу жолы. *«Ресторан» (ресторанға кіру) сала аумағына қатысты фреймдік білімдерді пайдалану үлгісін құру.* ***Шешу үдерісінің сипаты.*** Аталған фреймдік білімдерді пайдалану үлгісін құру үшін мынандай қадамдарды орындауымыз қажет:

1) мәселені шешуге қажетті сала аумағына қатысты абстракты объектілер мен ұғымдарды анықтау. Оларды фреймдер-түптұлғалар (фрейм-нысандар, фрейм-рөлдер) түрінде қалыптастыру.

2) сала аумағына қатысты нақты нысандарды анықтау. Оларды фреймдер-экземплярлар (фрейм-нысандар, фрейм-рөлдер) түрінде қалыптастыру.

3) салаға қатысты мүмкін болатын жағдайларды анықтау. Оларды фреймдер-жағдайлар (түптұлғалар) түрінде қалыптастыру. Егер сала аумағына қатысты прецеденттер болса, онда фрейм-экземплярларды (фреймдер-жағдайларды) қосу.

4) жағдайлар дамуының өзгеруін сипаттау. Оларды сахналар жиыны арқылы сипаттау. Оларды фреймдер-жағдайлар түрінде қалыптастыру.

5) нақты есепке қатысты деректерді бейнелейтін фрейм-нысандарды, сахналарды, қосу.

Мысалдың шешуі. Аталған сала аумағының басты ұғымдарына мыналар жатады: ресторан, ресторанға келетін адам (клиент), ресторанда қызмет атқаратын адамдар (аспазшылар, метрдотельдер, официанттар). Біз қарапайымдылық үшін тек официанттарды қарастырамыз. Қызмет ететін кісілер мен клиенттерді біріктіретін бір ұғым бар, ол – «Адам» ұғымы. Сонда «Ресторан» және «Адам» фреймдері түптұлғаүлгі фреймдері болады да, ал «Официант» және «Клиент» фреймдері түптұлға-рөл фреймдері болады. Сонымен бірге фреймдердің негізгі слоттарын да анықтау қажет. Яғни слоттардың осы шешілетін есепке қатысты қасиеттерін анықтап сипаттау қажет.

Фреймдер типтерін анықтауды ыңғайлы ету үшін, оны мынандай түрде бейнелейік:

< ***Фрейм Слот аты Слот мәні Мәнді алу тәсілі Демон*** >

Сонымен бірге мынаны атап өтуіміз қажет: слоттар мәнін «сыртқы көздерден» қабылдап алу деген ұғым бар. Бұл ұғым мынаны білдіреді: яғни қажетті ақпаратты қосымша (пайдаланушыдан, экраннан, құрылғыдан деген сияқты) ақпарат көздерінен аламыз. Сонда біздің мысалымызға қатысты фреймдер түрлері былайша бейнеленеді:

< ***АДАМ жынысы*** *Ер немесе әйел адам,* ***жасы*** *0 жастан бастап 120 жасқа дейін* >

< ***РЕСТОРАН Атауы Мекен-жайы Жұмыс істеу уақыты Бағыты*** *Класс Орташа* >

Фрейм-мұрагерлер өз аталарының барлық слоттарын өз құрамында ұстайды, олар белгілі бір параметрлер өзгерген жағдайда ғана анық түрде қолданыла алады.

< ***ОФИЦИАНТ*** *(АКО АДАМ)* ***жасы*** *18 жастан бастап 55-ке дейін;* ***жұмыс өтілі, жалақысы, жұмыс кестесі, жұмыс орны*** *Фрейм-нысан* >

< ***КЛИЕНТ*** *(АКО АДАМ)* ***Төлем түрі*** *Қолма-қол ақша немесе карточка Үнсіз келісім бойынша (Қолма-қол ақша)* ***Мәртебесі*** *Әдеттегідей немесе Vip Үнсіз келісім бойынша (Әдеттегідей)* ***Тапсырыс формасы*** *Тапсырыс бар немесе жоқ Үнсіз келісім бойынша (тапсырыс жоқ)* ***Сарқыттөлем*** >

Фрейм-үлгілер нақты жағдайды сипаттайды: қалада қандай рестorандар бар, оған бару қалай ұйымдастырылады, барушы кім, таңдап алған ресторанда кім жұмыс істейді, тағы басқа. Сондықтан біз өз мысалымызға қатысты келесі фрейм-үлгілерді анықтап алайық. Аталған фреймдер фрейм-түптұлғаларының мұрагерлері болады.

< ***КАФЕ-РЕСТОРАН «ДОС»*** *(АКО* ***РЕСТОРАН****)* ***Атауы*** *Дос* ***Мекен-жайы*** *Алматы, Мыңбаев көшесі, 15* ***Жұмыс істеу уақыты*** *9:00-00:00* ***Бағыты*** *Пиццерия* ***Класс*** *Орташа немесе Жоғарғы* >

< ***КАФЕ «МАГ»*** *(АКО* ***РЕСТОРАН****)* ***Атауы*** *Маг* ***Мекен-жайы*** *Алматы, Қонаев көшесі, 55* ***Жұмыс істеу уақыты*** *9:00-00:00* ***Бағыты*** *Паб* ***Класс*** *Орташа*>

< ***СЕРІК*** *(АКО* ***ОФИЦИАНТ****)* ***жасы*** *27* ***жынысы*** *ер адам* ***жұмыс өтілі*** *5* ***жалақысы*** *7 000* ***жұмыс кестесі*** *Күнара 18:00-ден 00:00-ге дейін* ***жұмыс орны «МАГ» КАФЕСІ*** >

< ***АЙЖАН*** *(АКО* ***ОФИЦИАНТ****)* ***жасы*** *24,* ***жынысы*** *әйел адам* ***жұмыс өтілі*** *2* ***жалақысы*** *8 200* ***жұмыс кестесі*** *Күнде сағат 9:00-дан 14:00-ге дейін* ***жұмыс орны «ДОС» КАФЕ-РЕСТОРАНЫ*** >

< ***БОЛАТ*** *(АКО* ***КЛИЕНТ****)* ***жынысы*** *ер адам* ***жасы*** *19* ***Төлем түрі*** *Қолма-қол ақша Үнсіз келісім бойынша (Қолма-қол ақша)* ***Мәртебесі*** *Әдеттегідей Үнсіз келісім бойынша (Әдеттегідей)* ***Тапсырыс формасы*** *Тапсырыс жоқ Үнсіз келісім бойынша (Тапсырыс жоқ)* ***Сарқыттөлем*** *тапсырыс мөлшерінен 7%* >

Фрейм-жағдайлар мүмкін болатын жағдайларды сипаттайды. Мысалы, ол тапсырыс пен оны төлейтін төлемдерге байланысты болуы да мүмкін. Ресторан жағдайында клиент кез келген түрлі уақиғаларға ұшырауы мүмкін: клиентке жасалған тамақ сапасы ұнамауы мүмкін, клиенттің тапсырысты орындауға ақшасы жетпей қалуы да мүмкін, тағы басқа жағдайлар. Осындай жағдайлардың кейбіреуіне мысал келтірейік. Олар, әрине, өмірде тіпті көп болуы мүмкін.

< ***ТАПСЫРЫС Тамақ мәзірі*** *IF-ADDED («Бағалар тізімі» слотын өзгертеді)* ***Бағалар тізімі Қосақталған процедура*** *IF-ADDED («Тапсырыс қосындысы» слотын өзгертеді)* ***Тапсырыс қосындысы*** *Қосақ-талған процедура* ***Тапсырысты қабылдады*** *Фрейм-үлгі* сыртқы көздерден ***Тапсырыс жасады*** *Фрейм-үлгі* сыртқы көздерден >

< ***ТӨЛЕМ Төлем түрі*** сыртқы көздерден *IF-ADDED («Сарқыттөлем» слотын өзгертеді)*

Сарқыттөлем *Қосақталған процедура* ***Төледі*** *Фрейм-үлгі қосақталған процедура*

Тапсырыс *Фрейм-образец* сыртқы көздерден *IF-ADDED («Төледі» слотын өзгертеді)* >

Әдетте, жағдайлар белгілі бір уақиғалардан соң және белгілі бір шарттар орындалған соң пайда болады да, бірінен соң бірі орындала бастайды. Сала мәселесіне қатысты осындай өзгеру динамикасын фрейм-сценарийлер арқылы сипаттауға болады. Олар өте көп болуы мүмкін, біздің мысалға қатысты кейбіреуін қарастырайық:

< ***РЕСТОРАНҒА КЕЛУ Келуші*** *Фрейм-нысан* сыртқы көздер-ден ***Ресторан*** *Фрейм-нысан* сыртқы көздерден *IF-ADDED, IF-REMOVED («Официант» слотын өзгертеді)* ***Официант*** *Фрейм-нысан Қосақталған процедура (таңдап алынған ресторан бойынша анықталады) Көрініс 1 Кіру, столды таңдау* сыртқы көздерден *Көрініс 2 Тапсырыс* сыртқы көздерден *Көрініс 3 Тамақ* сыртқы көздерден *Көрініс 4 Төлем* сыртқы көздерден *Көрініс 5 Шығу* сыртқы көздерден >

Біздің есепке қатысты Болат деген адам «Маг» ресторанына келді дейік. Сонда фреймдер былайша толтырылады:

< ***КЕЛУ «МАГ»*** *(АКО* ***КЕЛУ РЕСТОРАНҒА****) Келуші* ***БОЛАТ*** сыртқы көздерден *Ресторан* ***КАФЕ «МАГ»*** сыртқы көздерден *IF-ADDED, IFREMOVED («Официант» слотын өзгертеді) Официант* ***СЕРІК*** *қосақталған процедура (таңдалған ресторан бойынша анықтайды) Көрініс 1 Кіру, столды таңдау* сыртқы көздерден *Көрініс 2* ***БОЛАТ ТАПСЫРЫСЫ*** сыртқы көздерден *Көрініс 3 Тамақ* сыртқы көздерден *Көрініс 4* ***БОЛАТ ТӨЛЕМІ*** сыртқы көздерден *Көрініс 5 Шығу* сыртқы көздерден >

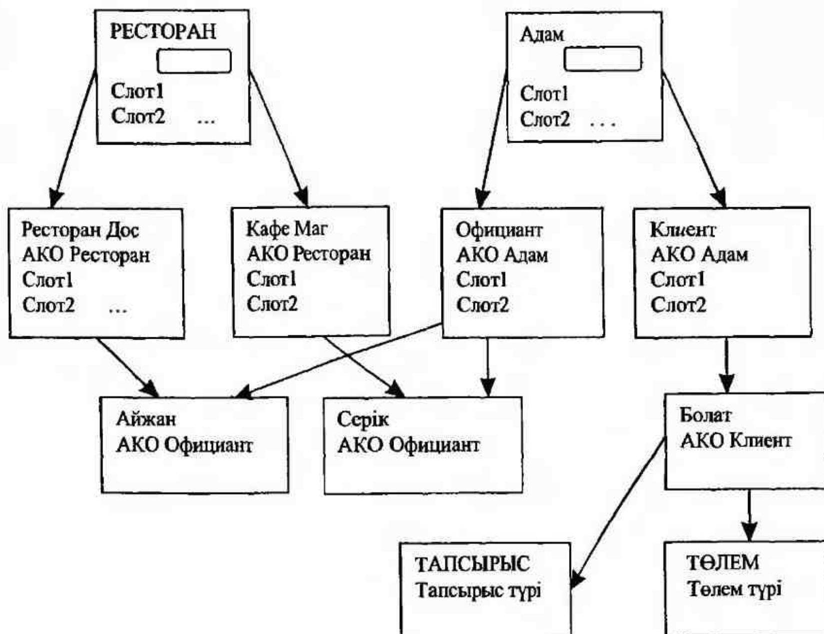
< ***БОЛАТ ТАПСЫРЫСЫ*** *(АКО* ***ТАПСЫРЫС****) Тамақтар тізімі Манты, алма шырыны* сыртқы көздерден *IF-ADDED («Бағалар тізімі» слотын өзгертеді) Бағалар тізімі 250, 75 қосақталған процедура IF-ADDED («Тапсырыс қосындысы» слотын өзгертеді) Тапсырыс қосындысы 325 Қосақталған процедура Тапсырысты қабылдады* ***СЕРІК*** сыртқы көздерден *Тапсырыс берді* ***БОЛАТ*** сыртқы көздерден >

< ***БОЛАТ ТӨЛЕМІ*** *(АКО* ***ТӨЛЕМ****) Төлем түрі Қолма-қол ақша* сыртқы көздерден *IF-ADDED («Сарқыттөлем» слотын өзгертеді) Сарқыттөлем 30 Қосақталған процедура* ***БОЛАТ*** *Төледі* сыртқы көздерден *Тапсырыс* ***БОЛАТ ТАПСЫРЫСЫ*** сыртқы көздерден *IF-ADDED («Төледі» слотын өзгертеді)* >

Осы мысалдағы фреймдердің әр түрлерінің арасындағы өзара байланыстар графикалық түрде граф секілді құрылыммен беріледі (4.6.6 суретін қара).

Бұл мысалда фрейм-көрсетім есебінде мына фреймдер: ***«Ресторан»*** және ***«Адам»*** жүреді. Ал фрейм-рөл есебінде ***«Ресторан Дос», «Кафе Маг», «Официант», «Клиент», «Серік», «Айжан», «Болат»*** жүреді. Ал фрейм-жағдай есебінде мына фреймдер: ***«Тапсырыс», «Төлем»*** жүреді. Сонымен бірге соңғы фреймдер өз құрамында қосақталған процедураларды ұстайды.

Фреймдік үлгіде жауап алу кезінде фрейм төбесімен қоса оның слоттары да есептеледі. Мысалы, «Маг» ресторанында официант болып кім жұмыс істейді?» деген сұраққа жауап алу үшін мына әрекеттер орындалуы қажет: сұрату мазмұнынан түсінікті болғандай, «Ресторан Маг» деген фреймді іздейміз және оның «Серік» фреймімен байланысын анықтаймыз. Соңғы фрейм «Официант» фреймінің мұрагері болып табылады. Сонымен бірге «Жұмыс орны» деген слотты да іздеп, оның фреймдердегі мәндерін іздейміз.



4.6.6-сурет. «Ресторан» саласына қатысты фреймдер сызбанұсқасы

Фреймдерде мынандай байланыстар анықталуы мүмкін:

< ***Клиент*** *АКО Адам* ***Официант*** *АКО Адам* ***Айжан*** *АКО Официант* ***Серік*** *АКО Официант АКО* ***Болат*** *АКО АКО Клиент* >

< ***АдамРесторан*** *Кафе-ресторан* ***«Дос»*** *АКО Ресторан Кафе* ***«Маг»*** *АКО Ресторан АКО АКО* >

< ***Ресторанға келу «Маг»*** *келу АКО Слот: келуші адам Слот: жұмыс орны Слот: жұмыс орны Слот: ресторан Слот: официант* >

< ***Тапсырыс Төлем Болат*** *Слот: көрініс 2* >

< ***Тапсырыс Болат*** *Слот: көрініс 4 АКО* ***Төлем*** *АКО Слот: төледі Слот: жасады Слот: қабылдады* >.

Бұл мысалда «Официант» фреймінің мұрагері болып «СЕРІК» фреймі табылады және ол «Маг» кафесінде жұмыс істейді.

Сарапшы жүйенің бөліктерін фреймдік үлгі негізінде құруға арналған жаттығуларды меңгеруге қажетті терминдерге мыналарды жатқызамыз: ***фрейм, слоттар, фреймдер типтері, слоттарды алу тәсілдері, демондар, граф түріндегі жазба, қосақталған процедура.***

Сарапшы жүйені фреймдік үлгімен жобалауға арналған тапсырмалар:

1. «Кітапхана» деп аталатын салаға қатысты фреймдік үлгіні жобалау.
2. «Аэропорт» (диспетчер бөлімі) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
3. «Теміржол» (билет сату) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
4. «Сауда орталығы» (ұйым) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
5. «Жанармай құю» (клиенттерге қызмет) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
6. «Автопарк» (жолаушы тасу) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
7. «Компьютерлік желілер» (ұйым) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
8. «Университет» (оқу үдерісі) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
9. «Компьютерлік қауіпсіздік» (қамтамасыз етудің құралдары мен тәсілдері) деп аталатын салаға қатысты фреймдік үлгіні жобалау.

10. «Интернет-кафе» (ұйым және қызмет) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
11. «Ақпараттық жүйелерді құру» (ақпараттық жобаны жүргізу) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
12. «Туристік агенттік» (клиенттермен жұмыс) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
13. «Зоопарк» (ұйым) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
14. «Қазанаспа» (тамақ дайындау) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
15. «Аурухана» (ауруларды қабылдау) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
16. «Кинопрокат» (ассортимент және клиенттермен жұмыс) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
17. «Автомобиль прокаты» (ассортимент және клиенттермен жұмыс) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
18. «Операциялық жүйелер» (қызмет ету) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
19. «Ақпараттық жүйелер» (түрлері және қызметі) деп аталатын салаға қатысты фреймдік үлгіні жобалау.
20. «Кәсіпорын» (құрылымы және қызметі) деп аталатын салаға қатысты фреймдік үлгіні жобалау.

4-бөлімнің бақылау сұрақтары

1. Білімдерді бейнелеу және қалыптастыру мәселесі.
2. Қосымша және көмек беруші білімдер айырмашылығы
3. Негізгі продукциялық жүйелер.
4. Көмек беруші білімдер түрлері
5. Талданатын білімдер типі
6. Конструктивті білімдер түрлері
7. Тұжырымдамалық граф түріндегі фреймдер ұғымдары.
8. Тұжырымдамалық граф түріндегі семантикалық торлар.
9. СЖ қызмет ету режимдері.
10. Фреймдегі иерархиялық құрылымдар.
11. Фреймдегі базалық типтер.
12. СЖ-дегі логикалық қорытынды шығару тәсілдері.
13. Іске асырылған СЖ типтері.
14. СЖ-ні жобалау талаптарын бағалау белгілері.

15. СЖ-ні қолдану аумағы.
16. СЖ-ні жобалау кезеңдері.
17. Білімді толықтыру әдістері.
18. Білімді толықтыру технологиясы.
19. Білімді ала білу тәсілдерінің жүйеленуі.
20. Білімдердің түрлері және формалары.
21. Семантикалық торлардағы шығару тәсілдері.
22. Табиғи тілді түсінудегі тұжырымдық үлгі.
23. СЖ қолдану аумағының шектеулері.
24. СЖ-нің құралдарының типтері.
25. Бірінші буын СЖ-нің құрылымы.
26. Білімдерді пайдалану үлгілеріндегі білімдер типі.
27. СЖ-нің қабықшалары.
28. Сарапшы жүйені пайдаланушылардың санаттары, қызметі.
29. Сарапшы жүйе – кеңес беруші режимдері.
30. Сарапшы жүйе – жобалау кезеңдері.
31. Продукциялық үлгінің артықшылығы мен кемшілігі.
32. Сарапшы жүйе құралдарының топтары.
33. Білім инженериясы тілдеріне қандай тілдер жатады?
34. Сарапшы жүйе қабықшаларына не жатады?
35. Гибридті сарапшы жүйе құралдары қандай?

5-бөлім. ҮЙРЕТУ ЖҮЙЕЛЕРІ

Компьютер жадында білім формасындағы ақпаратты жинақтап сақтау мәселесі білімді пайдалану техникасын меңгеруді ғана емес, оны толықтырып өзгертіп тұру техникасын да қолдануды қажет етеді. Бұл мәселе оқытып-үйрету мәселесімен бірте қайнасып жатады [9].

Оқытып-үйрету мәселесінің бірнеше аспектілері бар: үйренуші функциялары, ақпарат көзі – орта қасиеті, үйренуші мен ортаның өзара әсері.

Үйрену функциясы тірі табиғаттағы барлық тірі жандарға тән. Сондықтан осы жандарға тән, адамдар мен жануарлардың үйренуін бақылау және оны үлгілеу, математикалық немесе таңбалы логика көмегімен үйрену үлгілерін зерттеу, іс жүзінде болатын бейнелерді өңдегендегі үдерістерді үлгілеу сияқты мәселелер үйрету функциясын зерттегенде қарастырылады. Үйрену мәселесін үш түрлі бағытта қарастыруға болады. Бірінші – адамның ойлау мен үйрену функцияларын зерттеу және оны компьютерде үлгілеу. Екіншісі – ақпараттық логикалық үйрену функцияларын зерттеу. Үшінші – бейнелерді өңдеу аумағындағы үйрену үдерісін зерттеу. Болашақта бұл үш зерттеу бағыты бірігіп, бір аумақтағы сала мәселесіне қолданылып шешілсе, компьютерлік үйренуде белгілі бір жетістіктерге жетуге болар еді. Осы аталған бағыттардың үйрену функцияларын зерттеу бағытына жасанды нейрон желілерінің функцияларын зерттеуді жатқызуға болады.

5.1. Жасанды нейрондық жүйелер

Жасанды нейрон желілері өздерінің биологиялық ұқсастықтары нәтижесінде пайда болып, дамыды. Олардың қазіргі кезде шешетін мәселелер аумағы өте ауқымды. Олар бейнені тану, ұқсастыру, болжау, оңтайландыру, күрделі нысандарды басқару сияқты мәселелерді шешуде таптырмайтын құрал болып отыр. Ғалымдар дүние жүзіндегі аса зор жылдамдықты компьютерлердің дамуын қазірдің өзінде нейрокомпьютерлермен байланыстырып отыр. Ал олардың бәрі өздерінің негіздерін осы жасанды нейрон желілері теориясынан алады.

5.1.1. Жасанды нейрондық желілер тарихы

Жасанды нейрон желілері туралы ең алғашқы зерттеулер 1940 жылдары басталды. Нейрон желілерінің теориясы ғылыми бағыт ретінде

1943 жылы *У.Маккаллох* және *У.Питтс* деген ғалымдардың классикалық еңбектерінен бастау алады. Онда кез келген арифметикалық немесе логикалық қарапайым нейрон желісі көмегімен іске асыруға болатыны көрсетілген. Осындай іргелі зерттеулер жұмысына *Д.Хэбб* үлгісін де жатқызуға болады. Ол 1949 жылы жасанды нейрон желілерін үйрету алгоритмдерінің басы болып саналатын үйрету заңын ұсынды. Сонымен қатар нейрон желілерінің іргелі зерттеулер қатарына *М.Минский* теоремасын да, осы кісі зерттеген көпшілікке таныс «НЕМЕСЕ»ні алып «тастаушы» есебіне қатысты зерттеулерді де жатқызамыз [6].

Одан ары қарайғы зерттеулерге көз жүгіртсек, 1958 жылы *Ф.Розенблатт* ұсынған нейрон желісін атауға болады. Ол оны персептрон деп атады [34] және «Марк-1» деп аталатын бірінші нейрокомпьютерді құрды. Персептрон нысандарды жүйелеуге арналған және ол үйрену кезеңінде «оқытушыдан» оған ұсынған нысанның қандай топқа жататыны туралы мағлұмат алып тұрады. Үйретілген персептрон кез келген нысанды жүйелей алатын және ол үйрету кезінде пайдаланылмаған да нысандарды танып білу қасиетіне ие болатын.

1959 жылы *Д.Хьюбел* және *Т.Визель* атты ғалымдар биологиялық нейрон желілеріндегі ақпаратты өңдеу және сактау үдерістерінің үлестіргіш және қатар сипатта болатынын көрсетті.

Содан соң 1968-1985 жылдары нейрон желілері зерттеулерінде аздаған үзіліс болды. 1985-1986 жылдары нейрон желілері теориясы өзіндік «технологиялық серпін» алды. Ол сол кездерде пайда бола бастаған көпшілікке кеңінен тарап жатқан өнімділігі шапшаң дербес компьютерлердің дамуы еді [43].

1970-1976 жылдары КСРО-да персептронды зерттеулерге деген қызығушылық пен белсенділік әскери ұйымдарда болды.

1982-1985 жылдары *Дж.Хопфилд* ассоциативті жадыны үлгілейтін нейрондық желілер топтамасын ұсынды.

1985 жылы бірінші коммерциялық нейрокомпьютерлер пайда болды. Мысалы, АҚШ-тың TRW фирмасының MARK-3 деген нейрокомпьютерін атауға болады.

1987 жылдан бастап АҚШ, Жапония, Батыс Еуропа мемлекеттері нейрожелілер зерттеулеріне қаржылық демеулерін арттыра бастады. Осы жылдары Жапонияда «Human Frontiers» жобасы және европалық «Basic Research in Adaptive Intelligence and Neurocomputing» жобалары дүниеге келді.

1989 жылы дүние жүзіндегі көптеген ірі фирмалардың көпшілігі жасанды нейрон желілері мен нейрокомпьютерлерге қатысты зерттеулермен айналыса бастайды. Нейрокомпьютерлер ең сұранысы көп тауарлар қатарына ене бастады. Осыған байланысты АҚШ-тың қорғаныс министрлігінің қолдауымен DARPA агенттігі аса жоғары жылдамдықтағы нейрокомпьютерлерді құратын зерттеулерді қаржыландырды.

Осы кезде КСРО-да да нейрон желілеріне деген қызығушылық туып, осы мәселемен айналысатын фирмалардың саны үш жүзге жақындады. Нейрокомпьютерлер орталықтары Мәскеуде, Киевте, Минскіде, Новосибирскіде, Санкт-Петербургта құрыла бастады [50].

Осы кезде нейрон желілері мамандарының кеңінен пайдаланатын кітабы ретінде ғалым *Ф.Уоссерменнің* «Нейрокомпьютерная техника» деген зерттеуі болды. Ол орыс тілінде 1992 жылы басылып шықты [8].

1996-1997 жылдары жасанды нейрон желілері мен нейрокомпьютерлерге арналған дүниежүзілік конференциялар саны жүзге дейін жетті. Ал осы кезеңде нейрон желілері саласының зерттеулеріне бөлінетін қаржының жылдық мөлшері 2 млрд. доллардан асып түсті.

2000 жылдардан бастап, молекулярлық және биомолекулярлық технологиялар жетістіктерімен қоса, субмикрондық және нанатехнологияларға өту нейрокомпьютерлерді құруда жаңа архитектуралық және технологиялық шешімдерге қол жеткізуге көп мүмкіндіктер берді.

Жасанды нейрон желілерінің шешетін мәселелері де әрқилы болып келеді. Солардың кейбіреуін атап өтейік. *Бейнелерді жүйелеу (классификация образов).* Белгілер векторымен берілген ену бейнесін белгілі бір топтарға жатуын анықтау. Бұл мәселе бойынша шешілетін қолданбаларға: әріптерді тану, сөйлеуді тану, электрокардиограмма сигналдарын жүйелеу, кан клеткаларын жүйелеу сияқты мәселелерді жатқызамыз. Кластерлеу (категоризация). Бұл мәселе бейнені танудың оқытушысыз үйрету әдісіне жатады. Бұл топтау есептеріндегі алгоритмдер бейне ұқсастығына негізделіп, ұқсас бейнелерді бір кластерге топтайды. Шешетін есептер қатарына: білімдерді ала білу, деректерді қысу, деректер қасиеттерін зерттеу мәселелері жатады. *Функцияны аппроксимациялау.* Мысалы, мынандай $((x_1, y_1), (x_2, y_2), \dots (x_n, y_n))$ таңдау бар делік. Оны шуы бар функциямен генерациялайық. Аппроксимациялау мақсаты осы функцияның бағасын анықтау болып табылады. *Болжау.* Мынандай $\{y(t_1), y(t_2), \dots, y(t_n)\}$ дискретті N тізбегі $t_1, t_2, \dots, t_n$ уақыт аралығында берілген дейік. Шешілетін есеп мынандай: белгілі

бір t_{n+1} уақыт аралығында $y(t_{n+1})$ мәнін болжау қажет. Мұндай болжау бизнестің, ғылымның, техниканың қойылған мәселелерін шешуге септігін тигізеді. *Оңтайландыру.* Көптеген сала мәселелеріне қатысты белгілі бір шектеулер жүйесін қанағаттандырып, мақсатты функцияның ең үлкен немесе ең кіші мәнін табу есептері болады. Нейрон желілері әдістері осы есептерді шешуде де өз көмегін бере алады. *Мазмұны бойынша анықталатын жады.* Әдеттегі фон Нейман есептеулерінде жадыға қатынау тек оның мекені бойынша орындалады. Ал мазмұны бойынша анықталатын жадыға қатынау, оны әдебиетте ассоциативті жады деп атайды, оның мазмұны бойынша өтеді. Бұндай қатынау ерекшелігі мазмұн толық болмаса да іске аса береді. Бұл болашақтағы ақпараттық-есептеу жүйелерінде қажет болатын аса маңызды қасиеттердің біріне жатады. *Басқару.* Мынандай жиынмен берілген $\{u(t),y(t)\}$ динамикалық жүйені қарастырайық. Мұндағы $u(t)$ енуші басқару әсері, ал $y(t)$ белгілі бір t уақыт аралығындағы жүйенің шығу элементі болып табылады. Эталондық үлгісі бар басқару жүйелері үшін басқарудың мақсаты эталондық үлгі көрсеткен жолмен қозғалуды қамтамасыз ететін енуші $u(t)$ әсерін есептеп шығару.

Осындай әртүрлі мәселелердің барлығын нейрон желілерінің әдістері қалай шешеді? Қалыптасуы қиын немесе белгілі бір қалыпқа көнбейтін есептерді шешкенде екі түрлі бағыттағы әдістер қолданылады. Оның бірі – сарапшы жүйелерге тән ережелерге сүйенген тәсілдер тобы. Олар «егер ..., онда ...» деген ережелерге (rule-based) сүйенеді. Бұл жағдайда берілген білімдерді дәлелдеуді қажет ететін теорема есебінде қарастырып, қорытындыны шығару тізбегі көмегімен алады. Бұл бағыттағы әдістерді қолдану үшін сала аумағына қатысты бүкіл заңдылықтар сипатын анықтап, бере білу қажет. Екінші бағыт әдістері сала мәселесіне қатысты мысалдар (case-based) жиынтығын пайдаланады. Қалған білімдерді жүйе оқып үйрену нәтижесінде өзі тудырып отырады. Осы екінші бағыт жолы жасанды нейрон желілері пайдаланатын жолдарға дөп келеді.

Қазіргі кезде мамандар болжауы бойынша нейрондық желілер мен нейрондық компьютерлерді жобалау бағытында технологиялық өсім болатыны айтылып жатыр. Соңғы кезде нейрон желілерін пайдаланудың жаңа мүмкіндіктері ашылуда және осы ғылыми бағытында өнеркәсіпке, ғылымға, техникаға қосылатын үлестің өте зор экономикалық маңызы бар [44].

Жалпы жағдайда жасанды нейрон желілерін зерттеу үшін, ғылымның көптеген басқа салаларындағы терең білімдер қажет. Ол салалар қатарына: нейрофизиология, психология, физика, басқару теориясы, есептеу теориясы, жасанды зерде проблемалары, стаистика/математика, бейнені тану, компьютерлік көру, қатар есептеулер, цифрлық және ұқсас аппараттық құралдар жатады. Сонымен бірге жасанды нейрон желілерінің зерттеулері де өз жағынан осы аталған ғылым салаларын байытып, толықтырып қана қоймай, олардың ары қарай дамуына қажетті құралдарды береді.

Дегенмен қазіргі ғылымның даму сатысында адам миының ақпаратты қалай қабылдап, оны толық қалай өңдейтіні туралы табылған деректер әлі де аз және мардымсыз болып отыр. Дүние жүзіндегі нейробиологтар нейрондық үлгілеуді 50 жылдан артық уақыт бойына зерттеп келеді. Әйткенмен ми жұмысының барлық бөліктеріндегі өтетін ақпаратты өңдеу үдерісі әлі де толық түсінікті болмай келеді. Сонымен бірге мидағы бірде-бір нейронның ақпаратты импульстар тізбегі ретінде жіберуде қандай коданың қалай пайдаланылатыны әзірге құпия болып тұр.

Болашақта нейрокомпьютерлік технологиялар саласындағы күрт даму нейрондық үлгілеуге қатысты жаңа ашылымдармен байланысты болады. Ол адам баласы ми аумағындағы оның қызмет ету жолының ең болмағанда бір құпиясын аша алса, онда ол оның басқа бөліктерінде де қандай жұмыс атқарылатыны туралы мағлұматты білер еді.

Ақпаратты өңдеудің биологиялық негізін білу – жасанды ми құру жолындағы жұмыстарды тездетіп, ғылыми технологиялық жобаларды әлі адам баласының түсіне де кірмейтін биіктерге көтеру болар еді. Онымен салыстырғанда, қазіргі адам баласының ғарыштарды зерттеудегі, ядролық физика, молекулярлық биология, гендік инженерия сияқты ғылымның алдыңғы саласындағы жетістіктері қатардағы жай жетістіктер болып қалуы мүмкін. Бұл айтылып отырған жоба экономикалық тиімділік берумен қатар, ең алдымен «ақылды» машиналарды құруға, соның нәтижесінде адамдар үшін қауіпті және ауыр жұмыстарды атқаруға мүмкіндік берер еді [8, 44, 50].

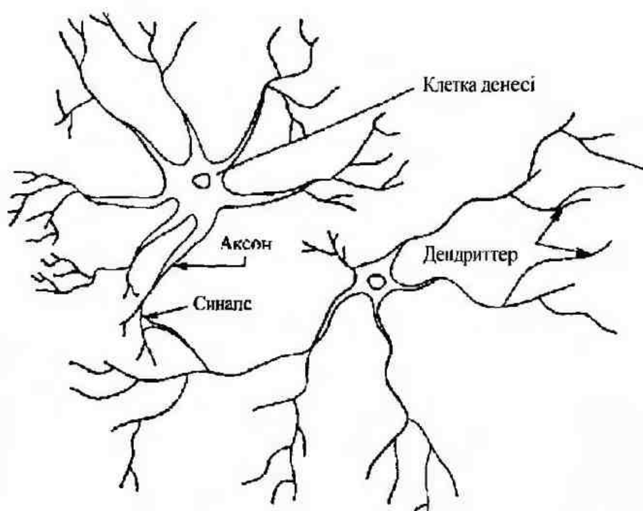
Сонымен бірге бұл мақсатқа жету үшін ғылымның ақпараттану, микроэлектроника, жасанды зерде осы сияқты тағы да басқа салалары қатар дамуы қажет.

5.1.2. Биологиялық нейрон

Зерттеулер тарихына көз жіберсек, *жасанды нейрон желілерінің* дамуына себеп болған, әрине, биология болды. Нейрон желілерін зерттеушілер желілік конфигурациялар мен алгоритмдерді тапқанда, яғни өз еңбектерінде ми жұмысының ұйымдастыру принциптерінен алынған терминдерін пайдаланды. Бірақ осы деңгейде ұқсастыру аяқталады десе де болады. Өйткені адам баласының нерв жүйесінің күрделілігіне тереңдеп бойлау – өте қиын мәселе. Нерв жүйесінің элементі болып табылатын **нейрон** – өте күрделі құрылым. Оның төңірегіндегі байланыстардың өзінің ұзындығы казіргі пайымдаулар бойынша метрден көп.

Әрбір *нейронның* өзінде көптеген қасеттер бар. Оның адам денесіндегі басқа органдардан ерекшелейтін негізгі касиеті – электро-химиялық сигналдарды нерв жолдарымен жіберу, қабылдау, өңдеу. Бұл нерв жолдары мида өзіндік коммуникациялық жүйе құрады.

Мына 5.1-суретте әдеттегі *нейрон* жұбының құрылымы көрсетілген. Осы суретте көрсетілген элементтер мен байланыс-тар жасанды нейрон желіеріне қатысты қарапайым қасиеттерді бейнелейді. Ол клетка денесінен (тело клетки – cell body), немесе оны әдебиетте сома (soma) деп атайды, және екі түрлі типтегі нерв талшықтарынан тұрады. Олар дентриттер (dendrites) және аксон-дар (axon) деп аталады. Дендриттер импульстерді қабылдайды, ал аксондар оларды жіберу қызметін атқарады. Сонымен бірге нейрон денесінде өзек (nucleus) бар. Онда тұқымның қасиетін сипаттайтын мағлұматтар болады. Ал нейрон денесіндегі плазмада нейрон даму-ына қажетті материалдарды өндіретін молекулярлық құралдар бар. Дендриттер нейрон клеткасы денесінен басқа нейрондарға жалғасып, синапс (synapses) деп аталатын косылуларда сигналдарды қабылдап алады. Бұл сигналдар аксон арқылы нерв талшықтарына (strands) тармақталады. Синапс қарапайым құрылым болып табылады, оның негізгі қызметі – ол екі нейрон арасында функционалдық түйін ретінде қызмет атқарады. Синапс қабылдап алған сигналдар нейрон денесіне келеді. Бұл жерде олар бір-бірімен қосылады. Түскен сигналдардың бірі нейронды қоздырса, екіншілері оның қозуына кедергі жасайды. Нейрон денесіндегі қоздырылған сигналдар қосындысы белгілі бір шектік шамадан артық болғанда, *нейрон* қозады да, аксон арқылы басқа нейрондарға сигнал жібереді.



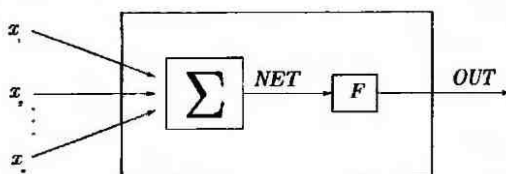
5.1-сурет. Нейрон жұбының құрылымы

Осы аталған үдерістерде болатын тәуелділіктер адам миындағы жады сияқты болуы мүмкін. Адам миында 10^{11} жуық нейрондар бар. Оның қалыңдығы 2-3 мм аралығында болады және оның аумағы 2200 $см^2$ құрайды. Әрбір нейрон саны 10^3-10^4 құрайтын нейрондармен байланыста болады. Жалпы жағдайда адам баласының осындай нейрон-аралық байланыстар саны мөлшермен 10^{14}-10^{15} құрайды.

Нейрондардың өзара әсері бірнеше милисекунд құрайды және олар импульстар серияларынан тұрады. Хабар жиілік-импульстік модуляция деп аталатын құрылым көмегімен беріледі. Жиілігі бірнеше бірліктен жүздеген герцке дейін болады. Әрине, бұл тез іске қосылатын электрондық сызбанұсқалардан миллион есе баяу. Соған қарамастан, күрделі тану есептерін адам бірнеше жүздеген милисекунд ішінде шеше алады. Ондағы операцияның орындалуы бірнеше милисекунд қана болады. Яғни есептеуде 100 тізбектелген кезеңдер болады. Басқаша айтқанда, күрделі есептер үшін адам миы 100 қадамнан тұратын қатар программаларды іске қосады. Яғни бұдан шығатын қорытынды – бір нейроннан екіншісіне жіберілетін ақпарат мөлшері өте аз болуы мүмкін, мысалы, бірнеше бит. Демек, нейрнондардағы ақпарат бірден жіберілмейді, ол нейрондар арасындағы байланыстарға үлестіріліп беріледі.

5.1.3. Жасанды нейрон құрылымы

Жасанды *нейрон,* негізінен, биологиялық нейрон қасиеттерін қайталайды. Жасанды нейронға енетін сигналдар жиынының әрбіреуі басқа нейронның шығу сигналы болып табылады. Әр ену сигналы синаптикалық күші бар салмақ шамасына көбейтіліп, осы көбейтінділердің барлығы бір-бірімен қосылады. Осы қосынды нейрон белсенділігінің деңгейін анықтайды.



5.2-сурет. Жасанды нейрон үлгісі

Осы негізгі ой желісін іске асырушы үлгінің сызбанұсқасы 5.2-суретте бейнеленген. Суретте көрсетілгендей, x_1, x_2, ..., x_n ену сигналдары жасанды нейронға келіп енеді. Бұл ену сигналдары жалпы жағдайда X векторының құрамына кіреді. Ол биологиялық нейронның синапстарына кіретін сигналдарға сәйкес келеді. Әр сигнал өзіне сай w_1, w_2, ..., w_n салмақтарға көбейтіліп, 5.2-суретте $\sum$ таңбасымен белгіленген қосушы блокқа келіп түседі. Нейронның математикалық үлгісін келтіре кетейік.

$$s = \sum w_i x_i + b,$$

$$y = f(s)$$

мұндағы w_i – синапс салмағы (weight), b – ығысу мәні (bias), s – қосу нәтижесі (sum), x_i – ену векторының бөлігі, i = 1, ... n, y – шығу сигналы, n – нейрондардың ену саны, f – сызықтық емес түрлендірулер (белсенділік функциясы).

Әр салмақ мәні биологиялык бір синаптикалық байланыстың «күшін» көрсетеді. Салмақтардың жиыны W векторы құрамына енеді. Биологиялық элементтердің денесіне сәйкес келетін қосу блогы салмақтарға көбейтілген енулерді алгебралық түрде қосады және соның нәтижесінде шығу сигналы алынады. Оны NET деп белгілейміз. Осы аталған операциялар векторлық қалыпта былайша белгіленеді:

$$NET = XW.$$

Одан соң NET сигналы белсенді F функциясы көмегімен түрленеді де, OUT нейрондық шығу сигналын береді. Белсенді функция ретінде қарапайым сызықтық функцияны алуға болады:

$OUT = K(NET)$.

мұндағы K – шектік функцияның тұрақтысы (константа). Нейрондық шығу сигналы функциясы мынандай мәндер қабылдауы мүмкін:

$OUT = 1$, егер $NET > T$,

$OUT = 0$ басқа жағдайларда,

мұндағы T – биологиялық нейронның сызықтық емес сипатын үлгілейтін кез келген тұрақты шектік шама немесе функция. Ол нейрондық желіге үлкен мүмкіндіктер береді.

Мысалы, 5.2-суреттегі F таңбасымен белгіленген блок NET сигналын қабылдап алып, OUT сигналын береді. Егер F блогы NET белгілі бір мәнінде OUT шамасын кез келген шектелген аралыққа орналастырса және осы жағдайда NET шамасының өзгеру диапазонын қысса, онда F функциясын **«қысу»** функциясы деп атайды. Мұндай «қысу» функциясы ретінде, әдетте, логистикалық немесе «сигмоидалдық» деп аталатын (S-бейнелі) функциясы қолданылады. Ол 5.2-суретте бейнеленген. Оның математикалық өрнегі былайша жазылады:

$F(x) = 1/(1 + e^{-x})$

Сонымен, шығу сигналы былайша өрнектеледі:

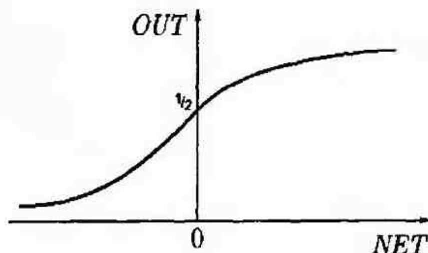
$OUT = 1/ (1 + e^{-NET})$

Электрондық жүйелерге ұқсас белсенді функцияны жасанды нейронның сызықтық емес түрдегі күшейткіш сипаты есебінде қарастыруға болады. Күшейткіш коэффициенті OUT шамасының түйірлік (приращение) қосындысының оны тудыратын NET шамасының түйірлік қосындысына қатынасы ретінде есептеледі. Ол белгілі бір қозу деңгейінде қисық сызықтың енкею дәрежесімен беріледі және күшті кері қозуда (қисық сызық көлденең жатады) аз ғана мәнде болып, нөлдік қозуда ең үлкен мәнге дейін қабылдайды. Егер қозу үлкен оң мағыналы болса, онда оның мәні тағы азаяды. Бұл сызықтық емес сипаттың сигналдың шуға қанығуын шешуге болысатынын ең бірінші рет 1973 жылы ғалым С.Гроссберг анықтаған. Бір желі түрі қалайша әлсіз сигналдарды және күшті сигналдарды өңдей алады? Пайдалануға ыңғайлы шығу сигналдарын алу үшін, әлсіз сигналдарға үлкен желілік күш салу қажет. Дегенмен үлкен күшейткіш коэффициенті бар каскадтар кез келген іске қосылған желідегі кездейсоқ флуктациялар сияқты күшейткіштермен араласып, шығу сигналдарының қаныққан түріне алып келуі мүмкін. Ал күшті ену сигналдары да күшейткіш каскадтардың қанығуына әкеліп, ол тағы да пайдалы шығу сигналын алуға мүмкіндік бермейді. Сонымен, үлкен күшейткіш коэффициенті бар логистикалық функцияның орталық аумағы әлсіз сигналдар-

ды өңдеуді шешсе, оң және теріс шеткі аумақтардағы төмендетілген күшейткіштер үлкен мәндегі қозулар үшін ыңғайлы болып келеді. Осылайша нейрон күшейткіштері ену сигналдары деңгейлерінің кең диапазонында жұмыс аткарады да, оның сигналы келесі формуламен анықталады:

$$OUT = 1/(1 + e^{-NET}) = F(NET)$$

Басқа кеңінен тараған белсенді функциялар қатарына гиперболалық тангенс жатады.

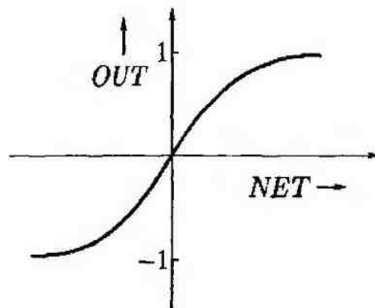


5.3-сурет. Белсенді функциялы жасанды нейрон

Оның сыртқы түрі логистикалық функцияға ұқсас болып келеді және, әдетте, оны биологтар нерв клеткасының белсенділігін көрсететін математикалық үлгі есебінде пайдаланады. *Жасанды нейрон желісінің* белсенді функциясы есебінде ол мына түрде өрнектеледі:

$$OUT = th(x).$$

Логистикалық функция сияқты гиперболалық тангенс S-ұқсас функция болып табылады, ол координат басымен салыстырғанда симметриялы болады және NET=0 нүктесінде OUT шығу сигналының мәні нөлге тең (5.4-суретті қара).



5.4-сурет. Гиперболалық тангенс функциясы

Гиперболалық тангенс функциясының логистикалық функциядан ерекшелігі – әртүрлі таңбалы мәндер қабылдайды. Сондықтан оның осы қасиетін көптеген желілерде пайдалануға болады.

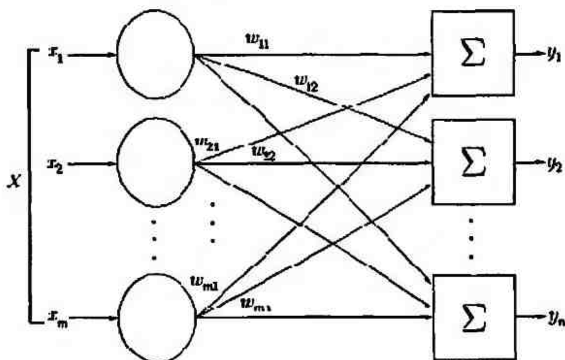
Нейронның белсенділік функцияларының түрлері бірнешеу. Оларға: экспоненциалдық, синусоидалық, сигмоидалық, қадамдық, шектік, модульдік, сигнатуралық, квадраттық түрлерін жатқызуға болады [41].

Қарастырылған жасанды нейронның қарапайым үлгісі өзінің биологиялық сыңарының көптеген қасиеттерін есепке алмайды. Мысалы, ол жүйе динамикасына әсер ететін уақыт аралығындағы тоқтауларды есепке алмайды. Ену сигналдары бірден шығу сигналдарын қалыптастырады. Тағы бір маңыздысы – жиілік модуляциясы функциясының әсеріне көңіл бөлмейді. Бұл биологиялық нейронның синхрондау функциясын зерттеушілер адам миындағы нерв қызметінде шешуші рөл атқарады деп есептейді.

Аталған шектеулерге қарамастан, осындай нейрондардан құрылған желі биологиялық жүйені еске салатын және оның қасиеттерін беретін желіні бере алады. Осы аталған ұқсастықтар кездейсоқ па әлде ол биологиялық нейронның ең маңызды қалыбының элементін бере алды ма деген сұраққа жауапты енді тек уақыт пен жасалатын ғылыми зерттеулер ғана бере алады.

Бір қатпарлы жасанды нейрон желілері.

Бір *нейронның* өзі қарапайым тану процедурасын іске асыруға мүмкіндік берсе де, нейрон есептеулерінің күші оларды желімен байланыстыруда жатыр. Қарапайым желі қатпаралар құрайтын нейрондар тобынан тұрады (5.5-суреттің оң жағындағы төртбұрыштарды қара).



5.5-сурет. Бір қатпарлы нейрондық желі

Сол жақта дөңгелекпен бейнеленген қатар қатпар емес, олар тек ену сигналдарын үлестіруге ыңғайлы болу үшін көрсетілген. Есептелуге жататын нейрондар төтбұрышпен белгіленген бөліктерде орналасады. Ену жиынының (X) әр элементі жасанды нейронмен жеке салмақ шамасымен байланысқан. Ал әр нейрон желіге енген кірулердің салмақталған қосындысын береді. Жасанды және биологиялық желілерде көптеген қосылулар түрлері берілмеуі де мүмкін. Тек қатпарлардағы ену және шығу элементтерінің қосылулары көрсетіледі.

Көрсетілген салмақтарды W матрицасының элементтері есебінде көрсету ыңғайлы. Матрицаның t жолы және n бағаны бар, мұндағы m – ену саны, ал n – *нейрондар* саны. Мысалы, w2,3 – екінші енуді үшінші нейронмен байланыстыратын салмақ. Сонымен, бір бөлігі OUT нейрондары болатын N шығу векторын есептеу үшін, мына матрицалық көбейтіндіні орындау қажет:

N = XW, мұндағы N және X – вектор-жолдар.

Көп қатпарлы жасанды нейрон желілері.

Әдетте, ірі және күрделі нейрондық желілердің есептеу мүмкіндіктері де үлкен болады. Адам баласының ой өрісі қаншаға жететіндей әртүрлі конфигурациялы желі түрлері құрылғанмен, *нейрондардың* қатпарлы түрде ұйымдастырулары мидың белгілі бір түрдегі құрылымдарын қайталайды. Соңғы кездегі зерттеулер нәтижелері мұндай *көп қатпарлы желілердің бір қатпарлы желілерге* қарағанда мүмкіндіктерінің зор екенін дәлелдеп берді. Қазіргі кезде осындай желілерді оқытуға арналған көптеген алгоритм түрлері бар. *Көп қатпарлы желілер* қатпарлар каскадынан құрылады. Бір қатпардың шығуы келесі қатпар үшін ену болып табылады. Желінің осындай түрі 5.6-суретте бейнеленген. Егер қатпарлар арасындағы белсенді функция сызықтық түрде болса, онда *көп қатпарлы желілердің* есептеу қуаты *бір қатпарлы желілермен* салыстырғанда көбейе коймайды. Қатпардың шығу элементін есептеу үшін, ену векторын бірінші салмақтық матрицаға көбейтеміз. Егер сызықтық немесе белсенді функция болмаса, онда алынған векторды екінші салмақтық матрицаға көбейтеміз.

(XW1)W2

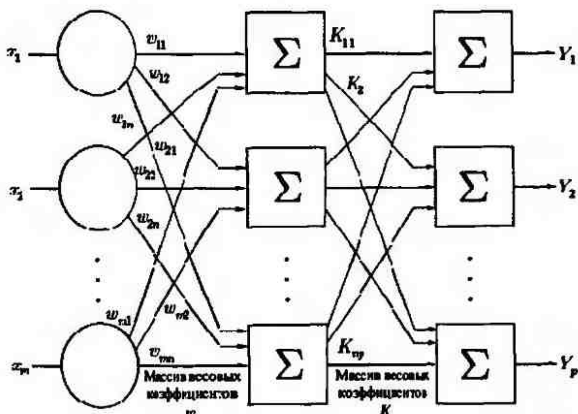
Матрицаларды көбейту ассоциативті болғандықтан:

X(W1W2)

Бұл формула екі қатпарлы сызықтық желінің бір-біріне көбейтілген екі салмақтық матрицасы бар бір қатпарлы желімен парапар екендігін

көрсетеді. Демек, кез келген сызықтық көп қатпарлы желіні бір қатпарлы желімен алмастыруға болады.

Бірақ *бір қатпарлы желілердің* есептеу қуаты аз болады. Сондықтан желі мүмкіндіктерін көбейту үшін, белсенді функцияның сызықтық емес түрі қажет.



5.6-сурет. Екі қатпарлы нейрондық желі

Осы кезге дейінгі біз қарастырған желілерде кері байланыс түрлері болған жоқ, яғни бір қатпардың шығуынан осы қатпардың өзіне немесе басқа қатпарға қайтадан кіретін байланыс түрлерін қарастырмаған едік. Бұндай байланыстары жоқ желі түрлерін арнайы топты желілер деп атайды. Немесе олар *кері байланысы жоқ* немесе *тура бағыттағы желілер* деп аталады. Ал шығудан енуге байланыстары бар желілерді *кері байланысы бар желілер* деп атаймыз. *Кері байланысы жоқ* желілердің арнайы жадысы-есі болмайды, олардың шығу элементтері толығынан ену сигналдары мен олардың салмактарына тікелей байланысты болады. Кейбір *кері байланысы жоқ* желілердің шығудың алдыңғы мәндері қайтадан кіру элементтеріне бара алады, сонда шығу элементтері ағымдағы енумен қоса оның алдындағы шығу элементтеріне де байланысты болады. Сондыктан *кері байланысы бар желілер* адам жадысындағы қысқа түрде еске сақтаумен ұқсас болады, яғни желілік шығулардың мәні оның алдындағы ену мәндеріне байланысты болады.

Желідегі қатпарлар санын есептеуге арналған арнайы тәсілдер әзірге жоқ. Әдетте, *көп қатпарлы желілер* өзара алмасып тұратын нейрондар

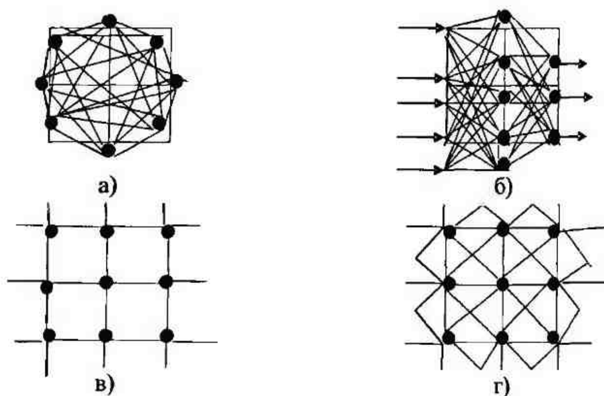
мен олардың салмақтарының жиынынан тұрады. Ену қатпарында қосу амалдары орындалмайды. Онда орналасқан нейрондар салмақтардың бірінші жиынының тармақталуын көрсету үшін ғана қажет. Олар желінің есептеу мүмкіндігіне ешқандай әсер етпейді.

5.1.4. Нейрон желілерінің жүйеленуі

Нейрон желілерінің атқаратын қызмет түрлеріне байланысты нейрондардың үш типін атап өтуге болады. Олар: *ену нейрондары*, ол сыртқы ортаның әсерін кодалайтын вектор, онда әдетте ешқандай есептеулер болмайды, ақпарат енуден шығуға қарай оның белсенділігін өзгерту арқылы жіберіледі; *шығу нейрондары* ондағы түрленулер нейронның математикалық үлгісі бойынша іске асырылады; *аралық нейрондар* нейрон желісінің негізін құрушылар түрлендіру нейронның математикалық үлгісі бойынша іске асады. Көптеген нейрондық үлгілердегі нейрон типтері оның желіде қалай орналасуына байланысты болады [43]. Топология жағынан нейрон желілерінің үш негізгі типтерін бөліп қарауға болады:

- толық байланған (5.7,а-суретті қара);
- көпқатпарлы немесе қатпарланған (5.7,б-суретті қара);
- байланысы әлсіз (5.7,в, г-суретті қара).

Бірінші типтегі нейрон желілерінде әр нейрон өзінің шығу сигналын кез келген желідегі нейронға бере алады. Көпқатпарлы нейрон желілерінде нейрондар қатпарларға топталады. Қатпардағы нейрондар саны кез келген болады және ол басқа қатпарлардағы нейрондар санына тәуелді емес.



5.7-сурет. Нейрон желілері архитектурасы

Ену қатпарын әдетте нөлдік деп белгілейді, ал соңғы шығу қатпарының сигналдары желінің шығу нейрондары болады. Сонымен бірге қатпарлы желілерде жасырын қатпарлар да болады. Көпқатпарлы желілердің өзі бірнеше түрге бөлінеді: *монотонды*, бұндай желілерде қатпардың екі түрі болады: қоздырушы, қоздырмаушы; *кері байланысы жоқ желілер* ақпарат ену қатарынан жасырын қатпарға өтіп, одан ары қарай шығу қатпарына дейін түрлендіруге түсіп беріліп отырады; *кері байланысы бар желілер*, соңғы қатпарлардың ақпараты қайтадан алдыңғыға беріле алады.

Байланысы әлсіз желілерде нейрондар төртбұрыш түйіндерінде орналасады (5.7,в, г-суретті қара). Әр нейрон өзінің 4 көршісімен (фон Нейман қоршауы), 6 көршісімен (Голей қоршауы) немесе сегіз көршісімен (Мур қоршауы) байланысқан.

Әдетте, нейрон желілерінің типтерін таңдау қойылған есеп түрі мен мақсатына тікелей байланысты болады.

5.1.5. Жасанды нейрон желілерін үйрету

Жасанды нейрон желілері қасиеттерінің ішіндегі ең ғажап қасиеттерінің бірі олардың үйренуге деген қабілеті болып табылады. Олардың бұл қасиеті адамның үйрену қасиетіне ұқсайтыны соншалық, тіпті адам баласы осы үдерісті тереңінен түсініп, оны қайталау мүмкіндігіне ие болды ма деген ойлар келуі де мүмкін. Бірақ ол әзірге тоқмейілсу болып қалуда. Өйткені *жасанды нейрон желілерінің* үйрену қасиеттерінің де белгілі бір шегі бар болып шықты. Ол адамның үйрену деңгейіне шығуы үшін, әлі де көптеген күрделі сұрақтарға жауап табу қажет.

Үйрену мақсаты. Желіні үйренді деп егер белгілі бір ену жиындарына шығу жиындары сәйкес келгенді айтамыз. Әрбір осындай ену (немесе шығу) жиындары вектор есебінде қарастырылады. Үйрену әрбір ену векторларын олардың салмақтарын әр кезде реттеу арқылы белгілі бір процедуралар көмегімен іске асады. Оқыту нәтижесінде желі салмақтары желідегі әрбір ену векторы өз шығу векторын тудыратындай дәрежеде түрленеді. Үйрету алгоритмдері *оқытушымен* және *оқытушысыз* деп бөлінеді.

Оқытушымен орындалатын алгоритмдерде әр ену векторы үшін қажетті шығуды сипаттайтын мақсатты вектор болады. Олар екеуі бірігіп, *үйренуші жұп* деп аталады. Әдетте, желі осындай үйренуші жұптардың бірнеше сандарымен жұмыс істейді. Желіге шығу векто-

ры ұсынылады, ол соған сәйкес мақсатты вектормен салыстырылып, алынған айырмашылық (қате) қайтадан кері байланыс көмегімен желіге жіберіледі. Содан соң алгоритм жолдарына сәйкес салмақтар өзгертіледі де, есептеу қайтадан жүргізіледі. Алгоритм қатені азайтуға тырысады. Үйрену векторларының жиынындағы векторлар желіге біртіндеп ұсынылады да, тағы қателер есептеледі, салмақтар әр векторға сәйкестендіріліп, тағы алгоритм өз жұмысын ары қарай жалғастырады. Осы үдеріс оқыту жиынындағы алынатын қате алдын ала қабылданған деңгейге жеткенге дейін орындалады.

Оқытушысыз үйрену. Көптеген жетістіктері бола тұрса да, оқытушымен үйрену тәсілі сынға ұшырады. Оған қойылатын негізгі кінә оның биологиялық тұрғыдан үйрену әдісіне ұқсас еместігі болды. Мида қажетті және нақты шығу сигналдарын өзара салыстырып, үйренуді кері байланыспен іске асыратын механизм бар екенін мойындау адамдар үшін қиын болды. Сондықтан биологиялық жүйелер үшін *оқытушысыз үйрену* адам қабылдайтын бейнелерге жақын болды. Бұл тәсілді Кохонен бастаған ғалымдар тобы зерттеп, одан ары қарай дамытты. Бұл тәсілде шығу сигналдарына арналған мақсатты вектор жоқ, онда алдын ала дайындалған жауаптармен салыстырулар болмайды. Үйрену жиынында тек ену векторлары ғана бар. Үйрету алгоритмі өзара келісілген шығу векторы пайда болғанға дейін желідегі салмақтарды жөндеп отырады. Яғни бір-біріне жақын ену векторлары өзара бірдей шығу векторын беруі керек. Демек, оқыту үдерісі оқыту жиынының статистикалық қасиетін бөлектеп алып, ұқсас векторларды топтайды. Осы топтағы ену векторы нақты анықталған шығу векторын береді. Бірақ үйренуге дейін қай шығу қандай ену векторлар тобымен іске асатынын болжау қиын. Сондықтан мұндай желінің шығуы оқыту үдерісіне сәйкес түсінікті түрге енуі қажет. Әдетте, ол оншама қиындық тудырмайды. Өйткені желі орнатқан ену мен шығу арасындағы байланысты анықтау аса қиын емес.

Үйрету алгоритмдері. Қазіргі кездегі үйрету алгоритмдерінің көпшілігі Д.О.Хэбб тұжырымдамасынан бастау алады. Ол ұсынған *оқыту-шысыз үйрету* үлгісінде егер екі нейрон да (жіберуші, қабылдаушы) белсенді болса, онда оның синапстық күші (салмағы) көбейеді. Осылайша желіде жиі пайдаланылатын жолдар күшейтіледі де, оқыту кезіндегі қалыптасу дағдысы мен қайталау арқылы үйрену жолдары адамға түсінікті түрде болып келеді. *Жасанды нейрон желілеріндегі* осы Хэбб тұжырымдамасы бойынша үйренуде

салмақтардың көбейтілуі жіберуші және қабылдаушы нейрондардың қозу деңгейлерін көбейту арқылы алынады. Бұл үйрену жолын былайша өрнектеуге болады: желі құруға дейінгі *нейроннан нейронға* өтетін салмақ мәні → желі құрудан кейінгі *нейроннан нейронға* өтетін салмақ мәні → оқыту жылдамдығының коэффициенті → *нейрон* шығуы және *нейрон* енуі → нейрон шығуы.

Хэбб тұжырымдамасы бойынша үйренуді пайдаланатын желілер біртіндеп даму үстінде болды. Дегенмен соңғы 20 жылдықта одан да тиімді үйрету алгоритмдері пайда бола бастады. Мысалы, оқытудағы ену бейнелерінің сипаттары кең диапазонды болып келетін және үйрену жылдамдығы қарапайым Хэбб тұжырымдамасы бойынша үйренуден тез болатын *оқытушымен үйрететін* алгоритмдер дамып, нейрон желілерінде қолданыла бастады.

5.1.6. Жасанды нейрон желілерін қолдану аумағы

Жасанды нейрон желілері кең түрде қолданылады. Нейрон желілерінің қолдану аумағына сөзді, сөйлемді, мәтінді, дауысты тани білу, мағынасы (семантика) бойынша іздеу, сарапшы жүйелер, шешімді қабылдау жүйелері, медициналық диагностика, акцияларды болжау, қауіпсіздік жүйелері, мәтіндерді талдау, қаржылық нарықтағы болжаулар, ақпаратты қысу жүйелері, техника және телекоммуникация саласы, Интернеттегі интеллектуалдық агенттер, жарнама және маркетинг, қатар нейроесептеулер сияқты көптеген салалар аумағын жатқызамыз [42].

Адам миының ең ғажап қасиеттерінің біріне оның бір нәрсені үйреніп алған соң, басқа бір жағдайларда үйренбеген әрекеттерді орындауы жатады. Үйренген жасанды нейрон жүйелері де белгілі бір сала аумағында бір рет үйренген соң, үйренбеген есептерін де шеше алады. Бұл жүйенің өзін-өзі үйрете алу қасиетіне жатады. Мысалы, қолмен жазылған көптеген әріптерді тани білетін жүйе осы әріптердің басқаша жазылуын да «танып» біле алады. Бұл жерде нейрон жүйесі өте тәжірибелі маман сарапшы адам сияқты әрекет жасайды. Мәселен, медициналық диагностика саласында «Нейропроект» компаниясы емшектегі балалардың есту қабілетін тексеру жүйесін құрды. Медицинада қабылданған жалпы әдістеме бойынша зерттеу үдерісінде берілетін белгілі бір дыбысқа мидың жауаптары тіркеледі. Олар электрэнцефалограммада шашылу биіктіктерімен сипатталады. Баланың есту қабілетін тексеру үшін тәжірибелі маман-аудиолог дәрігер 2

мыңнан артық тест жасайды. Ал жасанды нейрон жүйесі оны 200 бақылау нәтижесінде дәл осындай диагностика жасайды. Оған ол үшін бірнеше ғана минут жеткілікті.

Жасанды нейрон желілері қаржы нарығында да кеңінен қолданылады. Қазіргі кезде дүние жүзіндегі ірі банк жүйелері нейрондық желілерді өз қызметтерінде пайдаланып отыр. Мысалы, Chemical Bank деп аталатын банк транзакцияларды[1] алдын ала өңдеу арқылы бірқатар елдердегі өз валюта биржаларындағы күдікті келісімдерді тексеру мақсатында «Neural Data» фирмасы жасаған нейрожелілік жүйені пайдаланды. Осындай қоржындарды жүргізіп отыратын нейрожелілік автоматты жүйелерді «Deere & Co LBC Capital» компаниясы да қолданып отыр. Оның сарапшы жүйесі 900-ге жуық нейрожелілерді біріктіреді. Канаданың ірі банктерінің бірі CIBC банкі тәуекелділікті басқару үшін және қаскүнемдерді ұқсастырып табу үшін, өз жүйесінде «Angoss» фирмасының «KnowledgeSeeker» деген нейрожелілерді пайдаланатын программасын орнатты. Оның көмегімен банк мамандары болашақта банкке төлейтін өз төлемдерін қай клиенттің кешіктіретінін анықтағысы келді. Мамандардың жеке пікірлері бойынша ол клиенттер қатарына бұдан бұрын да төлемдерін кешіктіріп төлейтіндер кіруі керек еді. Бірақ нейрожелілер жасаған болжамда ондай клиеттерге өз төлемдерін уақытынан кешіктірмейтін, бірақ кей кезде «ұмытып» қалатындар жататындары келтірілді. Кейіннен дәлелденгендей, мұндай «ұмытшақтық» күрделі қаржылық қиындықтармен байланысты болды және нейрожелілердің болжамдарын растады.

Техника мен телекоммуникация саласындағы мысал ретінде NASA және «Air Force» тапсыруы бойынша жасалған автопилотты, гипердыбысты, LoFLYTE (lowObservale Flight Test Experiment) деп аталатын самолет-барлаушыны атауға болады. Самолеттің ұзындығы 2,5 м, салмағы 32 кг., болып ол ұшуды басқарудың жаңа принциптерін зерттеуге арналған болатын. Осы LoFLYTE самолетінде ұшқыштың ұшу әдістерін қайталау арқылы орындайтын автоұшқышты үйрету жолдарын нейрожелілер басқарды. Самолет жылдамдығы өте ұшқыр болғандықтан, ұшу режимінің өзгеруіне адам ұшқыштың жауап беру реакциясы төмен болды. Осы кезде көмекке нейрондық желілер келеді. Олар ақпаратты өте жылдам өңдеу нәтижесінде апатты және төтенше жағдайларда дұрыс шешім қабылдауды іске асыруға өз көмегін тигізіп

---

[1] Транзакция – ақпараттануда бірінен соң бірі орындалатын операциялар тобы. Транзакция толығымен орындалады немесе мүлде орындалмайды.

тұрды. Телекоммуникация саласындағы маңызды мәселелер қатарына тораптар арқылы ақпарат жіберуде оңтайлы жолды табу жатады. Бұл есепте де өте көп ақпараттар ағыны бар. Оған желідегі ағымдағы ақпараттар ағыны, байланыс сапасы, ақауға ұшыраған бөліктердегі ақпарат мөлшері сияқты кейбір мағлұматтарды келтірсек де жеткілікті. Осылардың бәрі – нақты уақыт көлемінде шешілуге тиісті жағдай. Бұл жерде де нейрондық желілер көмекке келеді. Сонымен бірге осындай күрделі телекоммуникациялық жүйелерді жобалағанда да шешімнің тиімді түрлерін алу үшін нейрондық желілер пайдаланылады.

Қазіргі кезде нейрожелілердің күрт дамуы олардың қатар нейроесептеулерде қолдануымен сәйкес келіп тұр. Арнайы нейрочиптер, кеңейту платалары көру, есту, бейнелер сияқты ақпарат түрлерін өңдеуге бағытталған. Ал осындай аппараттарға негізделген параллельдік есептеулердің маңызды бола бастауы осы механизмдердің іске қосу құралдарының дамуынан туып отыр десек, қателеспейміз.

Ақпараттық технологиялар саласында да жасанды нейрон желілері қолданылады. Мысалы, мәтіндік хабарлар тақырыптарын анықтау үшін, Интернеттегі жаңалықтарды жеткізудегі көшбасшылардың бірі PointCast компаниясы Convectis жаңалықтар серверін пайдаланады. Бұл сервер нақты уақыт аралығында тақырыпты олардағы негізгі сөздер мәнмәтін бойынша танып, автоматты түрде мәтіндік хабарлардың кең көлемін қазіргі кездегі алдыңғы қатарлы Reuters, NBC, CBS сияқты ақпараттық жүйелер арқылы таратуда. Тағы бір «Aptex Software Inc» компаниясының «SelectCast» деп аталатын нейрожелілік өнімі Ғаламтор пайдаланушыларының қызығушылық танытқан саласын анықтап, оларға осы тақырыптарға сәйкес жарнамаларды ұсынумен айналысады. Осы жүйені «Excite, Inc» компаниясы лицензиялап, өзінің іздеу серверлерінің жұмысында пайдалануда. Одан кейінгі жүргізілген зерттеулер осы жүйені Excite және Infoseek серверлеріне орналастырған соң, осындай тақырыптық жарнамаға деген қызығушылықтың бастапқыға қарағанда екі есе, ал кейбір түрлеріне бес есе артқанын көрсетті.

Нейрожелілердің араласқан тағы бір қызықты саласы – Интернет. Қазіргі ғаламтор заманында осы Интернеттің адамзат баласына тигізетін әсерін сөзбен айтып жеткізу қиын. Ғаламтордағы басты сөз, сөз тіркесі, сөйлем арқылы іздейтін іздеу жүйелері Интернеттегі желілермен берілетін өлшеусіз мәтін аумақтарын қопарумен бірге, онда талдаулар да жүргізеді. Бұл жерде де нейрожелілер алгоритмдерінсіз аяқ аттау қиын. Сол Ғаламтордағы зерделік-интеллектуалдық агент-

тер деп аталатын саланың болашағы зор. Бұл агенттер өз пайдаланушысымен ғана байланысып қоймай, өздері сияқты басқа агенттермен де байланысқа түседі. Бұндай байланыс түрлері арнайы сервис орталықтарына біріктіріледі.

Жарнама және маркетинг саласында да нейрондық желілер өзінің жеңісті көшін бастауда. Мысалға «Neural Innovation Ltd» компаниясының жұмысын алсақ. Ол бұрын өз қызметінде маркетингтік компаниялармен тура жіберу стратегиясында жұмыс атқаратын. Ол ең алдымен келіп түскен ұсыныстардың 25% ғана жіберіп, оған деген пайдаланушылардың пікірлерін жинайтын. Содан кейін осы деректер нейрондық желіге түсіп өңделгеннен кейін әр тауарға арналған сатып алу нарығының қажеттілігін анықтайтын. Осыдан кейін қалған 75% ұсыныстар нейрон желісі тапқан заңдылықтар бойынша көрсетілген сегменттерге жіберіліп отырды. Соның нәтижесінде компания түсіретін пайда бірнеше есе артты. Жалпы бизнесті жүргізуші компаниялар үшін пайдаланушылармен кері байланысты зерттеу өте қажетті жұмыстардың бірі болып табылады. Ол үшін компаниялар сатып алушылармен байланысты сауал арқылы жүргізеді. Бұндай сауал жүргізу нәтижелерін талдау күрделі есеп түріне жатады. Бұндай есептерде бір-бірімен байланысқан көптеген параметрлермен қоса, көп сұранысқа ие болатын тауарларды анықтайтын факторларды анықтау қажеттігі бар. Қазіргі кездегі нейрожелілік әдістер осы мәселені шешуге көмегін тигізіп қана қоймай, маркетингтік саясатты өзгерткенде, пайдаланушының қалай қимылдайтынына да болжам жасай алады [44-50]. Демек, нейрондық желілір компания жұмысының оңтайлы стратегиясын анықтауда таптырмас құралға айналады.

5.1.7. Жасанды нейрон желілерінің программалық өнімдері

Қазіргі кезде нақты мәселелерді жасанды нейрон желілері көмегімен шешуге арналған көптеген нейропакеттер бар.

Кейбір нейропакеттерге жүргізілген талдаулар нәтижесінде [41] олардың нақты қолданбалы есептерді шешуде белгілі бір жетістіктерге жететіні байқалды. Талдау кезінде жасанды нейрон желісінің әртүрлі бағалау белгілері бар көпқатпарлы архитектурасы, әртүрлі үйрету алгоритмдері таңдалып алынды. Осының нәтижесінде нейропакет мүмкіндіктерін көрсететін бірнеше көрсеткіш алынды. Бұл салыстыру көрсеткіштерін нейропакеттерді таңдаған кезде пайдалануға болады. Көрсеткіштерді атап өтейік:

- нейрон желісін құру мен үйрету қарапайымдылығы, түйсініп ұғуға жарайтын интерфейс;
- оқыту жиынының деректерін дайындау қарапайымдылығы;
- нейрон желісін құру және оқыту үдерісі кезіндегі ақпараттың айқындылығы мен толықтығы;
- нейрон желісін оқытуға қажетті алгоритмдер мен стандартты нейропарадигмалардың саны;
- жеке нейрондық құрылымдар құрудың мүмкіндігі;
- жеке оңтайландыруды бағалау белгілерін пайдалану мүмкіндігі;
- нейрон желісін оқытуға қажетті жеке алгоритмдерді пайдалану мүмкіндігі;
- нейропакет пен операциялық жүйенің басқа қолданбалары арасындағы ақпарат алмасуының қарапайымдылығы;
- архитектура ашықтығы, яғни нейропакетті жеке программалық модульдермен кеңейту мүмкіндігі;
- бастапқы коданы генерациялау мүмкіндігі;
- нейропакетпен жұмысты жылдамдататын макротіл болу мүмкіндігі.

Осындағы көрсетілген бірінші үш көрсеткіш нейропакетті жаңадан пайдаланушыға, 3-8 көрсеткіштер нақты қолданба есептерді шығаратын пайдаланушыға, ал 7-11 нейропакеттер негізінде жинақталған нейрон құралдар жүйесін жасайтын кәсіби маман программалаушылар мен жобалаушыларға қажетті көрсеткіштер тобын құрады.

Енді біз осындай нейропакеттердің кейбір түрлерін келтіре кетейік. Бұл пакеттер [42] көрсетілген. Нейропакеттер кестесі 1-қосымшада келтірілген.

Нейропакеттер келтірілген кестеде мынандай көрсеткіштер сипатталған: нейропакет атауы, Өндіруші, Платформасы, Үйрету алгоритмдерінің түрлері, интерфейс түрлері, түсіндірулер.

5.2. Үйрету әдістері

Бейнелерді өңдеу аумағындағы үйрену үдерісін зерттеу. Бұл үйренудің алдыңғы қарастырылған жасанды нейрон желілері ішіне кіретін зерттеулер болғанымен де, өзіндік ерекшеліктері бар. Сондықтан біз оны бөлек параграфпен қарап өтуді жөн деп санадық.

Бұл бағыттағы үйрету әдістерінің екі түрі бар. Ол *сұхбат* арқылы және *бейнені тану* арқылы үйрету әдістері.

5.2.1. Сұхбат арқылы үйрету

Үйрету тәсілдерінің біріне *когнитивтік тәсіл* деп аталатын әдіс жатады. Бұл тәсіл бойынша адамдар мен жануарлардың үйрену функцияларын үлгілеу тәсілдерін оқып-зерттеу үшін оларды бақылайды да, оның нәтижелерін талдайды. Үйрену функцияларын зерттеу компьютер көмегімен үйрету жүйелерін құру және жасау зерттеулерімен тікелей байланыста болады. Мұндай жүйе үлгілері табиғи тілді түсіну мен компьютерлермен сұхбат құруға байланысты құрамдас болып келеді. Бұл зерттеулердің ең қызықты саласы үйренуге қабілеттілікті зерттеу болып табылады. Үйренушінің үлгісі бақылау нәтижесінде анықталады [10].

Осы бағыттағы білімді ала біліп, үйрену үлгісінің мысалы ретінде диалог-сұхбат жүргізу арқылы болатын зерттеулерді қарастырайық.

Диалог пен үйрену арқылы білімдерді ала білу. Білімдерді диалог арқылы ала білу үшін ғылыми зерттеулерде «Мультииерархиялық үлгі» деп аталатын ұғым енгізілген. Бұл терминді түсіну үшін мынадай мысалды қарастырамыз. Адам мен ЭЕМ арасында келесі қарапайым түрдегі сұхбат бар делік.

Сұрақ: Қандай микрокомпьютерге *мына* принтерді қосуға болады?

Жауап: Анаған.

Сұрақ: Ол, *түрлі* басылым шығара ала ма?

Жауап: Иә.

Осы сұхбатта не жайлы әңгіме болып жатқанын білу үшін, микрокомпьютерлер мен принтерлер жайында техникалық білімдер мен *«мына», «анаған», «түрлі»* деген одағай сөздер жайлы лингвистикалық білімдер болуы керек [27].

Білімдерді ала білуде осындай сұхбат ұйымдастыру үшін мынадай қадамдар жасау қажет:

1) *салалық аумақты нақтылау үшін, шешілетін салаға қатысты білімдер көлемін шектеу қажет.* Жоғарыда келтірілген мысалға қатысты мынадай білімдер жиыны қажет: микрокомпьютердің аппараттық құралдары, программалық қамтамасы, оларды пайдалану тәсілдері жайлы білімдер, желілер, микрокомпьютерге қосылатын басқа аппараттар туралы, қызмет етуі жайында басқа да білімдер жиыны. Осындай білімдер жиыны анықталған соң, екінші қадамға өтеміз.

2) *білімдер не үшін керек?* Қазіргі кездегі зерделік жүйелер бес түрлі мақсатта пайдаланылады. Біріншісі фактілерді түсініп, талдауды түсінуге көмектеседі. Мысалы, микрокомпьютер жайында барлық сұраққа жауаптар алу. Бұл топқа әртүрлі кеңес беретін жүйелер мен

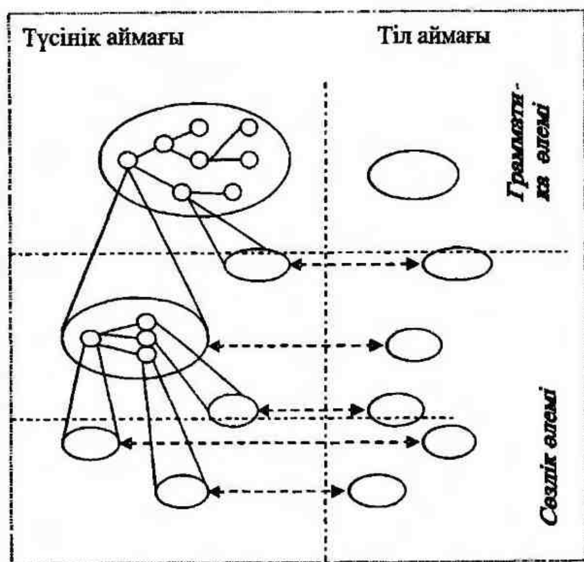
дерекқорға сұратулар жүйесі жатады. Екіншісі – мәселені шешу үшін. Бұл жағдайда негізгі проблемаға іздеу әдісі жатады. Бұл топқа жоспарлау жүйелері кіреді. Үшінші топта себеп пен салдарды салыстыру мәселесіне арналған жүйелер болады. Төртінші санатқа өзі сұрақ қойып, өзі жауап беретін жүйелер кіреді. Мысалы, компьютерлік үйрету жүйелерінде оқушы қатесін көрсететін бөліктер болады. Бесінші топқа жаңа білімді «қорытып», «түсіне» алатын, артық, қайшылығы бар білімдерді жүйелей алатын білімдер кіреді. Бірінші топтағы білімдерге білімдерді пайдалану үлгілерінің құрылымдарын қолданамыз. Екінші санат білімдерінің мәселесін шешу үдерісінің өзі үшін оның үлгілейтін функциясы болуы керек. Қалған топтағы білімдер құрылымдарында міндетті түрде мета-білімдер болуы қажет. (мета-білім дегеніміз – білімдер жайлы мағлұмат).

3) *білімдердің құрылымын жасау.* Белгілі бір салаға қатысты білімдердің құрылымын құруға арналған тәсілдердің біріне дүниенің иерархиялық үлгісін құру жатады. Мысалы: жоғарыда аталған микрокомпьютерлер туралы білімдерді алайық. Білім қорындағы білімдерді «*Желі* дегеніміз – компьютерлер желісі», «Дербес компьютер желінің бір бөлігі, ал программалық қамтама – дербес компьютердің бір бөлігі» деген сияқты сөйлемдермен берсек, осы сөйлемдердегі ұғымдардың бір-біріне қатынасу аумағында болу табиғатын «дегеніміз» (IS-A) деген қатынас түрі, «бір бөлігі болады» (PART-OF) деген, ал «элементтер болып табылады» (MEMBER-OF) деген қатынастар байланыстырып тұр.

4) *сыртқы дүниемен болатын интерфейс. Сыртқы дүние* дегеніміз – пайдаланушы адам дүниесі, яғни жүйеге қарағандағы сыртқы дүние. Бұл дүниемен байланыста болу үшін сыртқы дүниені бейнелейтін тіл (табиғи тіл, бейнелер) мен ішкі дүниені бейнелейтін (білімдерді пайдалану тілі), тіл арасында түрлендіру болуы керек. Осындай түрлендіру түрін *мультийерархиялық үлгі* деп аталатын құрылым бере алады (5.1-суретті қара).

Мультийерархиялық үлгі. Бұл үлгідегі интерфейс – *тілдер* аумағы, ал ішкі білімдер белгісі ұғымдар аумағы болып табылады [26]. Түсінік аумағындағы ұғымдар ең алдымен *грамматика* әлемінде талданады. Содан сон түсінік бөліміндегі ұғымдар бөлініп алынып (топ немесе жеке), *Сөздік* әлемінде талданады.

Осы үлгіге қатысты табиғи тілде сұхбат жүргізуді мына кезеңдерге бөлуге болады: *грамматикалық талдау, түсіну* және *қорытынды жасау,* сонымен бірге *синтез.* Грамматикалық талдау мен синтез – тілдер аумағындағы білімдер көмегімен, ал түсіну мен қорытынды жасау ұғымдар аумағындағы білімдер көмегімен іске асады.



5.2.1-сурет. Мультииерархиялық үлгі

Тіл аумағындағы сөйлем мағынасын түсіну былайша өтеді: сөйлемді грамматикалық талдау сәтті аяқталса, бірақ кейбір таңбалар тізбегін оқып-түсіну сәтсіз болса, онда «Қайталап анықтау» деп аталатын модуль іске қосылады [25]. Мысалы, ол сөйлемнің сәтті аяқталған бөлігінің түсінікті болғанын жариялап, мынадай хабар береді:

«Мен "қалдық" деген бөлікті түсінбей тұрмын», – деп, басқаруды келесі кезеңге береді. Егер ол да сәтсіз аяқталса, сөйлемнің мағынасын іздеу «басқа дүниеден» анықталады. Яғни «басқа дүние» есебінде басқа сөз қоры пайдаланылуы мүмкін.

Сұхбат жүргізу нәтижесінде анық емес сөздер, сөйлемдер кездесуі мүмкін. Мұндай сөздердің не сөйлемдердің мағынасын түсіну үшін, мультийерархиялық үлгіде мынадай әдістер пайдаланады:

Білімдерді йерархиялық түрде ұйымдастыру. Егер көп мағыналы сөздер әртүрлі мағынада қолданылса, олар әртүрлі дүниеге жатады. Олар арасындағы байланысты иерархиялық түрде беруге болады. Яғни қажет білімдер жоғарғы деңгейде, ал әртүрлі дүниелер төменгі деңгейде болады.

Бірнеше нұсқаны зерттеу әдісі. Анық емес сөздерді немесе сөйлемді ұқсастыру үшін, бірнеше болжам ұсынуға болады. Мысалы, «Ермек аудитория терезесін сындырды» деген сөйлемді талдағанда, «терезе»

сөзі мен «аудитория» сөзінің арасындағы түсініксіздікті «Ермек терезе сындырды» деген сөйлем мен «Ермек аудиторияға кірді» деген екі нұсқа арқылы салыстырып түсінуге болады.

Айнымалыларды теңестіру әдісі. Түсініксіз сөйлемдер мен одағай сөздерді тіл аумағындағы айнымалы есебінде қарауға да болады. Мысалы, «Ермек терезені таспен сындырды» деген сөйлемді былайша түрлендіруге болады (Пролог тілінде) [33].

1. сындыру (*агент, терезе,* құрал);
2. сындыру (* агент, терезе, тас);
3. сындыру (Ермек, терезе, тас).

Мағынаны сұрақ қою арқылы іздеу әдісі. Жоғарыда айтылған үш әдіс арқылы сұхбат жүргізу нәтижесінде түсініксіздік қалса, онда қосымша сұрақтар қою арқылы мағынаны анықтауға болады.

Анық емес білімдерді еске сақтау әдісі. Егер X анық емес білім болса, онда ол білімді пайдалану түрін мүмкін болатын ықтималдықты қолдану арқылы да шешуге болады. Мысалы, ол ықтималдық мәні P(X) = 0,3. Ықтималдықты сенім көрсеткішіне ауыстырсақ, аталған білім сенімділігі 30% болады.

5.2.2. Бейнені тану арқылы үйрету

Бейнені тану арқылы үйрету жүйелерінің қатарына жануарлардың көру жүйесі мен оны үлгілейтін тәжірибелік жүйелерді жатқызамыз. Мысықтардың көру жүйесінің жүйкелік физиологиясын анықтауға қойылған тәжірибелер нәтижесінде ғалымдардың жүргізген зерттеулері мынаны көрсетті: көру үдерісі көздің торшасынан басталып, ми қабықшаларына дейін барады. Ол мидағы өтетін үдерістер жүйкелік талшықтардан тұратын көп қатпарлы желіде өтеді. Ал осы жүйке клеткалары танып-білетін бейнелер осы аталған желінің элементтері болып табылады. Бұл элементтер қатпарлар деңгейі жоғарылаған сайын күрделене түседі [9].

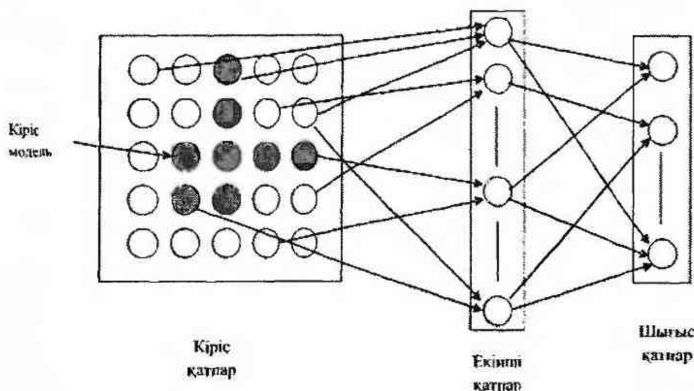
Сонымен қатар ғалымдар, жүйке физиологиясы жөніндегі белгілі білімдерге сүйене отырып, көрудің жүйкелік тізбегінің үлгісі деп аталатын құрылым құрды. Мұндай құрылым құру бейнені тану арқылы компьютерлік үйрену жүйесін зерттеудегі алға басқан маңызды қадам қатарына жатады. Енді осы үлгіге тоқталып өтелік [10].

Жүйкелік тізбелер желісінің үлгілері. Жоғарыда атап өткен ғалымдар зерттеген тәжірибелерде жануарлардың көруінің жүйкелік құрылымдары анатомиялық және психологиялық тұрғыда қарастырылып, түсіндірілген. Ал енді *Персептрон, Когнитрон, Неокогнитрон* деп ата-

латын жүйелерде осы аталған мәселерді конструктивті әдістермен шешуге әрекет жасалған. Яғни онда осылайша құрылған үлгілерді нағыз іс жүзіндегі нысандарға қолдану мүмкіндігі қаралады. Бұл мүмкіндік былайша іске асады: құрылған үлгі бойынша жұмыс істейтін көру жүйесінің жұмысы нақты тірі көру жүйесі функцияларымен салыстырылып, оның кеміс жақтары толықтырылады [34].

Неокогнитрон – мини компьютер негізінде іске асырылған тәжірибелік жүйе. Ол арнайы енгізу құрылғысы көмегімен жазылған таңбаларды тани біледі. Таңбаларды танып-үйрену үдерісі бірнеше секундқа созылады. Бұл жүйенің сыртқы белгі бойынша жүйелей білу қасиеті ғана емес, сонымен бірге оның үйрене білу қабілеті де бар. Яғни Неокогнитронға бір таңбаны бірнеше қайтара көрсетіп, жүйенің бұл таңбаны тани білу реакциясы тұрақты болғанға дейін және бұл үдерісті келесі тануға қажет таңбаға өткенде қайталаса, онда жүйе бейнелерді тани білу қабілетіне ие болады. Бұдан ары қарай үйрену үдерісіне кері байланыс тізбегін қоссақ, онда жүйе ассоциативтік түрде бейнелеу қабілетін үйренеді.

Персептрон – мұғалімдік үйрету жүйесі. 1958 жылы Розенблатт бейнелерді танып үйренудің, үш қатпарлы құрылымдық жүйкелік тізбелер үлгісіне негізделген жүйені құрды. Ол оны Персептрон деп атады, оның құрылымы 5.2.2-суретте көрсетілген.



5.2-сурет. Үш қатпарлы перцептронның құрылымы

Бұл жүйенің жұмыс істеу ерекшелігі мынада: бірінші – қатпардағы бірнеше элементтерден шығатын сызық байланыстар, екінші қатпардағы бір элементте тоғысуы мүмкін.

Құрылымды пайдалану ыңғайлылығы – екінші және үшінші қатпарлар элементтері арасындағы қатынастарды байланыс коэффициенті деп аталатын көрсеткіш арқылы басқару мүмкіндігі. Әр элементтің екі шығу мәні бар: 1 және 0. Жүйені бейнені тануға үйрету үшін бірнеше қатпарға сырттан бейне түрі беріледі де, ал үшінші қатпар онымен салыстырылатын бейне-реакция түрлерін беріп отырады. Ал *Персептрон*, берілген үйрету ережелеріне сай, байланыс коэффициенттерін қажетті реакция алынғанға дейін өзгертіп отырады. Сыртқы белгілер көрсеткіштері бойынша мұндай жүйе мұғаліммен біріге отырып үйренетін жүйе деп аталады.

Үйрену механизмі. Үйрену үдерісін қателерді жөндеуге назар аударылатын үдеріс деп атауға болады. Қателерді жөндеу байланыс коэффициенттерін түзету арқылы өтеді. Қате реакция болған кездегі түзеудің барлық байланыс коэффициенттеріне қатысты болады. Түзетудің көптеген ережелері сынақтан өтіп, оның барлығында да үйрету сәтті аяқталып отырған. Осындай ережелердің ең қарапайым түрі мынандай болып келеді: **i** – элементінің $\mathbf{x}_{ij}$ енгізілуі, оның шығуы $\mathbf{d}_{ij}$ және оның байланыс коэффициентінің Д коррекциясы арасындағы тәуелділік былайша анықталады:

егер $\mathbf{x}_{ij} = \mathbf{d}_{ij}$, онда **aij + Д**;

егер $\mathbf{x}_{ij} \neq \mathbf{d}_{ij}$, онда **aij – Д**, мұндағы $\mathbf{a}_{ij}$ – байланыс коэффициенті.

Тәжірибелік *Персептронда* 1-ші қатпарда 20 х 20 = 400 элемент, екіншісінде – 400, үшіншісінде 8 элемент болады. Тәжірибеде бұл құрылым белгілі бір орында көрсетілген белгілі бір әріптерді дұрыс тани білген.

Ішкі байланыс түрі анықталған жүйе. Алдыңғы құрылымға тағы бір қатпар қосамыз. Оның негізгі қызметі – бірінші және екінші қатпарлар арасындағы байланысты іске асырады. Төртінші қатпардың шығысы екі элементпен анықталады: *енгізу* жиыны мен оған көрсетілетін әсерлерді *білдіру* жиыны. Сондықтан бұл қатпарларда уақытша тәуелділіктер пайда болады. Яғни кейбір бейнелерді тануға үйрету үдерісі кезінде бейне бойында жүріп өткен қимыл түрлеріне арналған уақытша тізбектерді құру мүмкіндігі де беріледі. Осындай үлгілеу арқасында нақты іс жүзінде мынандай нәтижелер алынған: белгілі бір бейнеге әр кезде жүйе бір түрдегі жауап беріп отырған. Осы бейнені басқа жерге көшіріп, оны сол жерде айналдыру бейнесін еске сақтаған. Екінші бейнеде де болған өзгерістерді компьютерге үйретіп отырған. Яғни көшірілген және оны айналдырған кездегі өзгерістерді осындай үйрету нәтижесінде қозғалыс формасын (екі

өлшемді) компьютерге үйрету мәселесі қозғалған. Дегенмен қазіргі таңда компьютерді осындай жолмен үйрету тәсілі әлі теориялық түрде толық зерттелмеген.

Жүйке клеткаларының логикалық функциялары. Жүйке клеткалары Персептрон жүйесінің ең кішкене конструктивтік элементі есебінде жүреді. Бірінші қатпар клеткалары оларға жарық түсу не түспеу нәтижесінде қозады не қозбайды. Егер клетка қозса, онда **1** деген сигнал беріледі де, ол екінші катпарға жетеді. Екінші және одан кейінгі қатпарларда $\mathbf{x_i}$ енгізу сигналы алдыңғы қатпардың **i**-ші клеткасының қозуына байланысты екі мән қабылдайды. Осы қатпардағы **i**-ші мен **j**-ші клеткалар арасындағы $\mathbf{a_{ij}}$ байланыс коэффициенті және **i**-ші клетка мен оның шығу $\mathbf{y_i}$ элементінің $\mathbf{и_j}$ аралық-шектік мәні арасында болатын байланыстар былай бейнеленеді:

$$\sum (a_{ij} \cdot x_i) - и_j > 0 \qquad y_i = 1;$$
$$\sum (a_{ij} \cdot x_i) - и_j < 0 \qquad y_i = 0;$$

Мұндағы $x_i = 1$ немесе 0. Мұндай логикасы бар механизмді *аралық* логика деп атайды. *Аралық* логика теориясы толық зерттелген десе де болады. Қарапайым құрылымды аралық элементтерді *ЖӘНЕ, НЕМЕСЕ* деген логикалық элементтер арқылы үлгілеуге болады. Оған *ЕМЕС* деген терістеуді қоссақ та болады. Осы элементтер комбинациясы көмегімен кез келген логиканы іске асыруға болады. Сонымен қатар әр элементтің өз кешігу уақытын ескере отырып, еске сақтау функциясын да іске асыруға болады. Осыларды пайдалана отырып, жүйке тізбектері желісіне ұқсастырып, компьютердің арифметикалық-логикалық құрылымын жасауға болады.

Енді жеке клетканың логикалық функциясының қызметін геометриялық тұрғыдан қарастырайық. Егер клеткаға ену саны – **n**, ал шығу мәні тек **1** не **0** болса, онда ену бейнелерінің жиыны **n** өлшемді бірлік кубты құрады. Егер біз куб төбелеріне клетканы қоздыратын не қоздырмайтын бейнелерді орналастырсақ, онда бұл бейнелер жиынын екі кіші жиынға бөлуге болады. Ол бөлуді **n** өлшемді қиюмен іске асырамыз: $\sum (\mathbf{a_{ij}} \cdot \mathbf{x_i}) = \mathbf{и_j}$;

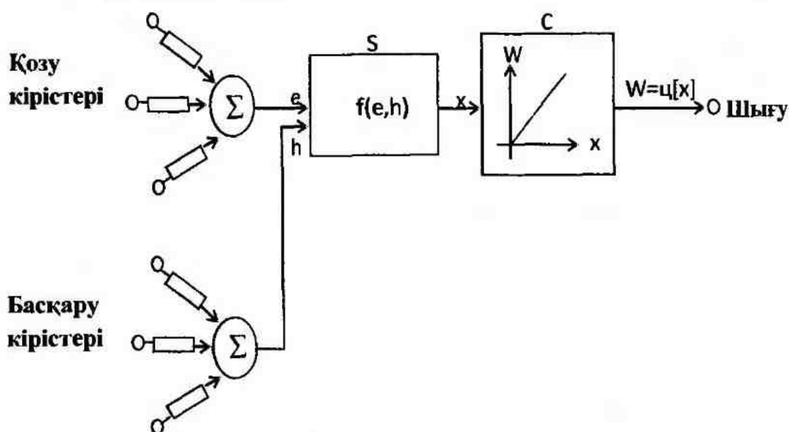
Персептрондағы үйрену процесінде клетка қозған жағдайда мәні өзгеріп, енгізілген бейнеге әсерін қиманың бұрылу бұрышынан байқаймыз.

Сонымен, жоғарыда аталған *Персептрон* мен *Неокогнитрон* деп аталатын құрылымдар компьютерді параметрлік түрде үйретуге қабілетті механизмдер қатарына жатады.

Неокогнитрон – «мұғалімсіз» үйрету жүйесі. Үш қатпарлы *Персептрон* жүйесінде пайдаланылатын үйрету әдісін «мұғаліммен бірге» үйрету жүйесі деп атайды. Бұл әдісте енгізу бейнесімен қатар, оны тану нәтижесі де көрсетіліп отырады. Керісінше, әр бейнені тану кезеңінде оған деген шығыс әсері жеке талданып отырса, оны *«мұғалімсіз»* үйрету тәсілі деп атаған.

Осындай тәсіл негізінде құрылған жүйкелік тізбектер желісінің бірнеше үлгісі ұсынылған. Сондай үлгілердің біріне *Неокогнитрон* жүйесі жатады. Бұл үлгідегі үйрету тәсілі де, үлгідегі негізгі элемент-жүйке клеткасының үлгісі де, жүйкелік клеткалар желісінің құрылымы да *Персептрон* үлгісінен өзгеше болады. *Неокогнитронның* негізгі өзгешелігі мынада:

- клетканың жауап әсері дискретті емес, яғни екі ғана мән қабылдаумен шектелмейді, жауап әсері үздіксіз түрде болады;
- үйрету принципі есебінде «ең үлкен мәнді табудың ғылыми болжамы» деп аталатын тәсіл пайдаланылады. Бұл тәсілдің негізгі сипаты мынадай: «Жергілікті түрде ең үлкен шығу әсерін көрсететін клеткалар сигналы көрсетіледі». Ондағы қолданылатын *Күшейту ережесі: «ең үлкен шығу әсерін көрсететін клеткалардың кіру сигналдарының байланыс коэффициенттері нөлге тең болмаса, онда ол кіру мәніне тең болатын шамаға арттырылады»;*
- бақылау клеткалары енгізіледі. Бұл клеткалардың байланыс коэффициенттері теріс мәнде болады.



5.2.3-сурет. Жүйке клеткасының үлгісі

Неокогнитронның Персептроннан тағы бір өзгешелігі ондағы қатпарлар санының көптігінде (5.2.3-суретті қара). Сонымен бірге өзгеріп тұратын байланыс коэффициенттері бар катпарлардан соң тұрақты байланыстағы қатпаралар орналасады. Тұрақты байланыстар белгілі бір S қатпарының бөліктерімен *НЕМЕСЕ* деген байланыс түрімен катынаста болады да, S бөлігінің бір элементінің қозуы С қатпарының элементтеріне әкеледі. Мұндағы X = f(e, h) белгілі бір S қатпарындағы және W = ц(x) С қатпарындағы жүргізілетін есептеулер функциялары, ал ∑ – белгісі қозу және басқару клеткаларының қосындылары [10].

Неокогнитрон авторлары (Фукусима мен Миякэ) [25,26] екі түрлі жүйенің түйіскен жерлеріне кері байланыс тізбектерін енгізу арқылы ассоциативтік функциялары бар автореакция үдерісін іске асыруға болатынын көрсете білді. Егер жүйе алдын ала үйренген бейнелерін бірден «үйме» түрінде оған көрсетсе, онда ол осы үймедегі барлық бейнелерді бірінен соң бірін көрсетіп бере алатын қабілеті барын білдіре алады. Бұл қабілетті авторлар тәжірибе жүзінде көрсеткен. Бұл *Неокогнитронның* ассоциативтік қасиетке ие екенін көрсетеді.

Коннектионист бағыты. Жоғарыда аталған *Неокогнитрон* және *Персептрон* жүйелері көру жүйесінің үлгілеріне жатады. Олар, негізінен, бейнені тану қызметін атқарады. Ал *Коннектионист* жүйесін құру бағытында бейнені тану қызметімен катар, таңбаны өңдеу қызметінің де бірге орындалып отыруы қарастырылған. Мұндай жүйелердегі бейнені тану үдерісі жүйкелік клеткаларға ұқсас элементтері бар құрылым арқылы өтеді. Ал таңбаларды өңдеу мәселесін әртүрлі базалық сызбанұсқалар жинағы атқара алады деген ой жүйесіне негізделген. Осындай сызбанұсқалар түрлеріне: іске асыру механизмі *ЖӘНЕ/НЕМЕСЕ* екі каскадтық сызбанұсқаға негізделген «Егер ... онда» деген ережелер жинағын, ең үлкен мәнді таңдау сызбанұсқалар жиынын, кестелерді қарау арқылы іске асыру сызбанұсқалар жинағы сияқты әр түрдегі тәсімдерді пайдалануға болады. Осындай сызбанұсқалардың әртүрлі комбинациясын жасау арқылы түрлі алгоритмдерді алуға болады. Яғни бұл бағыт зерттеулерінде *Персептрондағы* жоқ идеялар пайда болған.

Көріністер (Изображение) арқылы үйрену. Алдыңғы карастырылған үлгілер параметрлік үйрету әдістеріне жатады. Яғни олардағы нысандар ұғымдары накты физикалық нысандармен тікелей байланыста болып отырған. Егер үйрету кезінде нысанда кездесетін элементті ешқандай ұғым түріне жатқызу мүмкіндігі болмаса ше? Ал

ол элементті «мынаған ұқсас» деген ұғыммен белгілесек, онда бұл ұғымның өзін компьютер түсінігінде қалыптастыру мәселесі пайда болады. Осындай ұғымды қалыптастыруда Уинстонның үйрету теориясы зор рөл атқарады [36]. Уинстон ұғымды компьютерге үйретуге, сурет көмегімен алынатын графикалық көрсетуді қалыптастыру арқылы жасауға болатынын көрсетті. Ақпаратты өңдеу жүйелерінің болашағы осындай суреттер мен бейнелерді енгізу және шығару формаларынан тұратын жүйелер болады деген үміт бар. Қазіргі кездегі нысандарды тану үдерісі оның сыртқы түрлері арқылы тану бойынша іске асырылып отырған. Алдымен оның контуры, жазықтықтағы формалары анықталып, соңынан олардың ұғымдарымен сәйкестігі табылған. Кейінірек, бейнелерді өзара салыстыру жайындағы білімдер пайдаланыла басталған. Яғни төменнен жоғары қарай өңдеу, үлгілер бастамасы бойынша басқару деген сияқты.

Зат көрінісі негізінде берілген құрылым талдауы. Зат көрінісі негізінде берілген құрылым талдауының бірінші кезеңінде оны белгілі бір бөліктерге бөледі. Әр бөліктегі формалар мен олардың функциялары анықталады. Егер осындай бөлу мен формаларды анықтау сәтті аяқталса, екінші кезеңде абстрактылы ұғымның сипаты анықталған бөлік ұғымдарының өзара байланысы ретінде табылады. Яғни екінші кезеңді орындау аса қиындық тудырмайды. Жүргізілген ғылыми зерттеулерде бірінші кезеңде жатық жергілікті симметрия тәсілін пайдаланып, құрылым бөліктерін формасын анықтаған.

Бақылау сұрақтары:

1. Нейрондық желідегі қызмет кезеңдері.
2. Жасанды нейрон зерттеулерінің тарихы.
3. Биологиялық нейрон құрылымы.
4. Жасанды нейронның үлгісінде қандай элементтер бар?
5. Жасанды нейронның белсенді функциялары қандай?
6. Бір қатпарлы нейрон желісі.
7. Көпқатпарлы нейрон желісі.
8. Нейрон типтері.
9. Нейрон желілерінің түрлері.
10. Жасанды нейрон желісі шешетін есептер.
11. Жасанды зерде жүйесіндегі үйренудің бағыттарын атаңыз.
12. Үйретудегі когнитивтік тәсіл деген не?
13. Сұхбат тәсілінің қадамдары.

14. Мультийерархиялық үлгі деген не?
15. Мультийерархиялық үлгідегі қолданылатын әдістер.
16. Бейнені танудағы жүйелер үлгілері.
17. Неокогнитрон деген қандай жүйе?
18. Персептрон алгоритмі.
19. Персептрон жүйесіндегі жүйке клеткаларын үлгілеу.
20. Үйрету жүйелеріндегі Коннектионист бағыты.
21. Уинстонның үйрету теориясы туралы түсінік.
22. Жасанды нейрон желісі алгоритмдерінің қолдану аумағы.
23. Нейропакеттер ерекшеліктерін атаңыз.
24. Нейропакет мүмкіншіліктерін анықтайтын қандай көрсеткіштер?
25. Нейропакеттерде қолданылатын платформалар қандай?
26. Нейрожелілердің қолдану салаларын атаңыз.

6-бөлім. ТАБИҒИ ТІЛДІ ТҮСІНУ

Зерделік жүйелердегі табиғи тілді түсіну жолдары мәселе шешілетін аумақтағы білімдерге тікелей байланысты болады. Тілді түсіну деген ондағы сөздер мен сөйлемдерді айту ғана емес. Ол үшін аталған тілде сөйлеп тұрған адам сала мәселесінің түсінігін де сол тілде жеткізе білуі керек. Табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдердің анықталған пайдалану құрылымдарын қажет етеді. Осындай құрылым түрлерін жасағанда білімдердің өзгешелігін білдіретін қасиеттерімен бірге, адамдар арасында болатын күрделі қарым-қатынастар түрлерін, бір сөздің бірнеше мағынада айтылуы, үйрену әдістері, әртүрлі көзқарастар сияқты мәселелердің барлығын есепке алу қажет.

Табиғи тілді түсіну мәселесін шешу үшін бірнеше сұраққа жауап табу керек. Біріншіден, адам білімінің үлкен ауқымды көлемі қажет. Табиғи тілді түсінуді іске асыратын компьютерлік жүйеде күрделі де нақты дүниедегі шым-шытырық байланыс түрлері сипатталуы қажет. Екіншіден, табиғи тілдің өзінде белгілі бір құрылымдар бар: сөздер дыбыстар мен таңбалардан, ал сөйлемдер сөздерден тұрады. Дыбыстар мен таңбалардың, сөздердің орналасуы белгілі бір тәртіпке бағынады. Бұл тәртіпті сақтамаса тілді түсіну қиындайды. Тағы бір мағлұматтарды қамтитын сұрақтар түріне адам немесе компьютер (оларды әдебиетте агенттер деп атап жүр) тудырған өнім есебінде бағаланатын тілдік құрылым жатады.

Сарапшы жүйе түріндегі компьютер программалары пайдаланушы адаммен табиғи тілде сұхбат жүргізеді. Жасанды зерде теориясының зерттеулерінде табиғи тіл мәселесіне көп орын беріледі. Қазіргі кезде адамша сөйлей алатын программалар жоқ деп айтуға болмайды. Дегенмен белгілі тақырыпқа қарапайым сөйлемдер құрылымын енгізу арқылы сұхбат жүргізе алатын компьютерлік программалар жасалған. СЖ программаларындағы пайдаланушы мен сұхбат жүргізетін программалар, әдетте, мынандай қасиеттерімен сипатталуы мүмкін: компьютер сұхбатты базалық мағлұматты жинақтау үшін жүргізеді де, өзінің білім қорындағы сипатталған жағдайлармен жағдайды салыстырады. Яғни сұхбаттың сипаты мынадай болып келеді: кез келген түрде берілген қарапайым сөйлемдерден тұратын жауаптарды түсініп, қорыту қабілеті; білім қорындағы берілген жағдайларға сәйкес, программаның сұрақтар қоя білу қабілеті; қажет болғанда өзі келген қорытындысын табиғи тілде түсіндіре алу қабілеті.

Табиғи тілде сұхбат жүргізу әдістері. Табиғи тілде сұхбат жүргізуді мына кезеңдерге бөлуге болады: *грамматикалық талдау, түсіну* және *қорытынды жасау*, сонымен бірге *синтез* [9]. Грамматикалық талдау мен синтез тілдер аумағындағы білімдер көмегімен, ал түсіну мен қорытынды жасау ұғымдар аумағындағы білімдер көмегімен іске асады. Тіл аумағындағы сөйлем мағынасын түсіну былайша өтеді: сөйлемді грамматикалық талдау сәтті аяқталса, бірақ кейбір таңбалар тізбегін оқып-түсіну сәтсіз болса, онда белгілі бір модуль іске косылады. Сұхбат жүргізу нәтижесінде анық емес сөздер, сөйлемдер кездесуі мүмкін.

Тілді түсіну үшін оны талдауды таңбадан бастау қажет. Тіл күрделі ерекше құбылыс түріне жатады. Тілді талдау үшін онда келесі аталған үдерістерді зерттеу қажеттігі болады. Ондай үдерістерге: дыбыстар мен таңбалы әріптерді тану, синтаксистік талдау, жоғарғы деңгейдегі семантикаларды зерттеу жатады. Сонымен қатар осындай үдерістер қатарына: ритм және дыбыс ырғағы мен күші арқылы берілетін сөздердегі мағына түрлерін де жатқызады. Осындай күрделі үдерістерді басқару үшін, тіл мамандары табиғи тілді таңдаудың әртүрлі деңгейлерін анықтады. Осындай деңгейлердің: Просодия, Фонология, Морфология, Синтаксистік талдау, Семантика, Прагматика, Төңіректегі дүние жайлы білім деп аталатын жеті түрін атап өтуге болады [10].

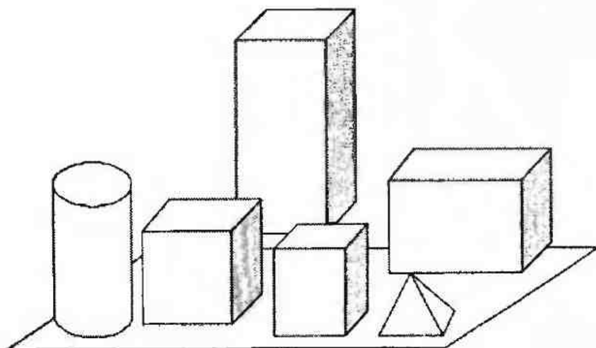
Зерделік жүйелердегі табиғи тілді түсіну жолдары мәселе шешілетін аумақтағы білімдерге тікелей байланысты болады. Тілді түсіну деген ондағы сөздер мен сөйлемдерді айту ғана емес. Ол үшін аталған тілде сөйлеп тұрған адам сала мәселесінің түсінігін де сол тілде жеткізе білуі керек. Қорыта айтсақ, табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдердің анықталған пайдалану құрылымдарын қажет етеді.

6.1. Табиғи тілді түсіну проблемалары

Жасанды зерденің ең алғашқы жүйелерін құрушылар программа жұмыс істейтін сала аумағын азайтуға тырысты. Осындай құрылған жүйелердің біріне SHRDLU (1972) жатады [9].

Жүйе мынандай қызмет атқарды: әртүрлі түрдегі және түстегі блоктармен жұмыс істеп, осы блоктарды «қол» көмегімен араластырып отырды. Осындай блоктардың мысалы 6.1-суретте келтірілген. SHRDLU жүйесі ағылшын тілінде қысқаша қойылған сұрақтарға жа-

уап беріп, қарапайым сұратуларды орындап отырды. Мысалы, мынандай сұрақ: «Қызыл блоктың үстінде не тұр?» немесе мынандай сұрату: «Жасыл пирамиданы қызыл блоктың үстіне қой». Жүйе қойылған сұрақты түсінумен бірге оған айтылған бұйрықты да орындап отырды. Мысалы: «Бұл жерде қызыл түсті блок бар ма? Соны көтер». Жүйе блоктардың түсі, формасы мен бірге оны «тани» білу қабілетіне де ие болды. Мысалы: «Көк түсті блоктың үстінде тұрған фигураның түсі қандай? Ол қандай блок?».



6.1-сурет. Блоктар «дүниесі»

Блоктар «дүниесі» қарапайым болғандықтан, жүйедегі білім қоры сала білімдерімен толық толтырылған. Бұл «дүниемен» қарым-қатынасқа түсу үшін, күрделі ұғымдар (уақыт, мүмкіндік, пікір сияқты) жайында білімдердің қажеті бола қоймайды. Қаралатын сала мәселесі белгілі түрде шектелген ұғымдардан тұратындықтан, SHRDLU жүйесінде синтаксис пен семантика бірігуі сәтті шыққан [9]. Осы жүйенің мысалы мынаны көрсетеді: сала мәселесі туралы жеткілікті білім көлемі болса, онда құрылатын компьютерлік жүйедегі табиғи тіл мәселесін шешу оңайға түсер еді.

Табиғи тілді түсіну мәселесінің тағы бір қамтитын жағына сала аумағындағы жеткілікті білімдермен бірге таңдап алынған табиғи тілдің шаблондық үлгісі мен түсінікті құрылымдары да қажет. Осындай құрылымдардың біріне «марков тізбектері» – «марковские цепи» – «markov chain» деп аталатын құрылымды жатқызуға болады. Мысалы, тіл сөйлемдерінде сын есімдер мен одағай сөздер («анау кісі», «сұлу қыз») әдетте зат есім алдында тұрады. Марков үлгілері осындай қарым-қатынас түрлерін бейнелеуге ыңғайлы болып келеді.

6.2. Таңбалы талдау

Тіл табиғатын түсіну үшін, оны талдауды таңбадан бастау қажет. Кез келген табиғи тілдің өз ерекшелігі бар. Тіл ерекшелігіне қатысты құбылыстарға: дыбыстар, таңбалар, әріптер, сөздерді құру табиғаты, мән-мағынаны беретін семантиканы жатқызады. Тілді талдау үшін ондағы болатын үдерістерді зерттеу қажеттігі болады. Осындай күрделі үдерістерді басқару үшін, тіл мамандары табиғи тілді таңдаудың әртүрлі деңгейлерін анықтады. Зерттеулер осындай деңгейлердің жеті түрі бар екенін көрсетті [9].

1) ***Просодия (prosody).*** Бұл деңгейде тілдің ритмі мен дыбыс ырғағы және күші арқылы берілетін тілдің мағынасы талданады. Бұл деңгейдегі зерттеулер әлі толық емес. Сондықтан қазіргі жобаларда бұл мәселені айналып өтеді десе де болады. Ритм мен дауыс ырғағына бағынатын мәтіндер поэзияда, діни білімдерде кездеседі. Тіл ритмикасы халық әуендері мен бесік жырларында көп.

2) ***Фонология (phonology).*** Тілдік құрылымдардағы дыбыстар мен олардың әр түрде байланысқан түрлерін зерттейтін сала. Тілдің бұл саласы компьютерлік танып білу мен сөйлеуді қалыптастыру мәселесімен айналысады.

3) ***Морфология (Morphology).*** Сөздер морфем деп аталатын бөліктерден тұрады. Тілдің бұл саласы осы морфемдерді талдауға арналған. Сонымен бірге бұл салаға сөздерді қалыптастыру ережелері, қосымша мен суффикстерді пайдалану ережелері кіреді. Морфологиялық талдау сөйлемдегі сөздердің мағынасын анықтауда, етістіктер уақыттары мен сөз бөліктерінің тәртібін де тексеруде де қолданылады.

4) ***Синтаксистік талдау (Syntax).*** Әртүрлі сөздер мен сөз құрылымдарындағы сөздердің жұптасу ережелерін зерттейді. Осы ережелердің сөздер мен сөйлемдер мағынасын талдаудағы атқаратын қызметін де бағалайды. Тілдің бұл саласының қалыпқа келу үдерісі көбірек зерттелген, сондықтан ол лингвистикалық талдауды автоматтандыруда кеңінен қолданылуда.

5) ***Семантика (Semantics).*** Сөздердің, сөз құрылымдарының, сөйлемдердің мағынасын зерттеп, оның табиғи тілде бейнелеу әдістерін зерттейді.

6) ***Прагматика (Pragmatics).*** Тілді пайдалану және оның тыңдаушыға әсерінің әдістері туралы ғылым.

7) ***Төңіректегі дүние туралы білім (Word knowledge).*** Адам баласы өмір сүріп отырған орта жайында, адамдар арасындағы әлеуметтік қарым-қатынас жөніндегі білімдер. Бұл жалпыға қатысты білімдердің табиғи тіл мен бейнеленген мәтіндер немесе сұхбаттардағы сөйлемдерді түсінудегі атқаратын рөлі зор.

Табиғи тілдегі жоғарыда аталған деңгейлердің барлығы бір-бірімен тығыз байланыста болады. Мысалы, «Ол алма жеп отыр» деген сөйлемнің өзін әр деңгейде әртүрлі талдау түріне саламыз. Әсіресе синтаксис пен семантика арасындағы айырмашылықты компьютер зердесіне салу үшін, белгілі бір құрылымдар мен олардың үлгілері қажет.

Қорыта айтсақ, табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдердің анықталған пайдалану құрылымдарын қажет етеді. Осындай күрделі үдерістерді басқару үшін, тіл мамандары табиғи тілді таңдаудың әртүрлі деңгейлерін анықтады. Осындай деңгейлердің: *Просодия, Фонология, Морфология, Синтаксистік талдау, Семантика, Прагматика, Төңіректегі дүние туралы білім* деп аталатын жеті түріне шолу жасадық. Осы аталған жеті деңгейдің алдыңғы үшеуіне арналған ізденістер өте аз болғандықтан, қалған төрт деңгейге қатысты әдебиеттерде кездесетін және қолданыста бар зерттеулерге шолу жасайық.

Синтаксистік талдау. Синтаксисттік талдаудың негізін грамматикалық талдау құрады. Ол үшін қазіргі кездегі зерттеулерде [9, 10] шығару ережелері /rewrite rule/ деп аталатын ұғым енгізілген. Бұл ережелер грамматиканы анықтау мақсатында қолданылады. Мысалы, «Адам итті жақсы көреді», «Ит тістейді» деген сөйлем түрлерін қарастырайық. Бұл сөйлемді талдау үшін әдебиеттерде қабылданған ереже түрін келтірейік.

1. sentence<->noun_ phrase verb_phrase,
2. noun_phrase<-> noun,
3. noun_phrase<->article noun,
4. verb_phrase<-> verb,
5. verb_phrase<-> verb noun_phrase,
6. article <-> ол,
7. article <-> мына,
8. noun<-> адам,
9. noun<-> ит,
10. verb<-> қабады,
11. verb<-> ұнатады.

Ережелер ретпен нөмірленген. Мұндағы белгілер мынаны білдіреді:

Sentence – *сөйлем*
Noun_phrase – *заттық құрылым*
verb_phrase – *етістік құрылым*
noun – *зат есім*
verb – *етістік*
article – *одағай сөз*

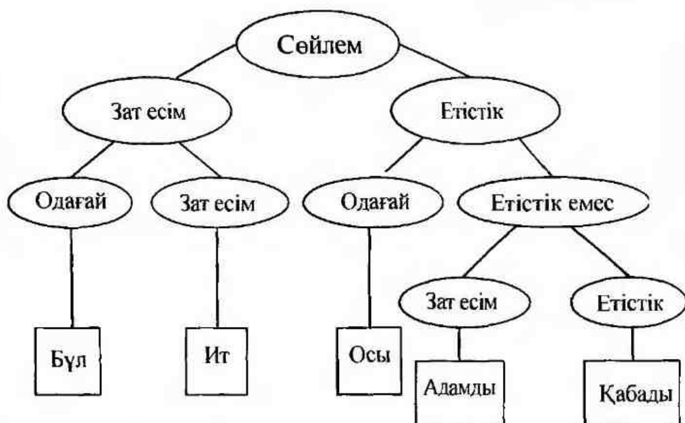
6-11 ережелердің оң жағында берілетін сөздер табиғи тіл сөздерін құрайды. Бұл ережелер сала мәселесіне қатысты сөйлемдерде пайда болатын сөздерден тұратын сөз қорын құрады. Бұл сөздер грамматика терминалы деп аталып, сөздік қор құрады. Жоғарғы деңгейдегі лингвистикалық ұғымдарды (Sentence, Noun_phrase) анықтайтын терминдер терминалдық емес түрлері деп аталып, олар формула стильдерімен бөлінеді (6.1-кестені қара).

Терминалдар ережелердің сол жағында кездеспейді. Дұрыс анықталған сөйлем түріне жоғарыда аталған ережелер көмегімен бөлуге болатын кез келген терминалдар қатарын айтамыз. Сөйлем өзгерісі, яғни трансформациясы терминалды емес Sentence таңбасынан басталып, грамматика ережелерін сақтап орындалған көптеген бірінен соң бірі келетін ауыстырулардан кейін терминалдар қатарын қалыптастыруға әкеледі. Бұл сөздер грамматика терминалдары (terminal) деп аталып, сөз қорын (lexicon) құрады. Мұндай көрсетімді сентенциалдық форма (sentential form) деп атайды.

6.1-кесте

Қатар	Ереже нөмірі
sentence	1 *сөйлем*
noun_ phrase verb_ phrase	3 *зат есім – етістік*
Article noun verb_ phrase	7 *одағай сөз – етістік*
The noun verb_ phrase	8 *зат есім – етістік*
The «адам» verb_ phrase	5
The «адам» noun verb_ phrase	11
The «адам ұнатады» noun _ phrase	3
The «адамды қабады» article noun	7
The «а дамды қабады мына ит»	9

«Осы адамды бұл ит қабады» деген сөйлемнің ережелер бойынша түрленуі келесі 6.2-суретте келтірілген:



6.2-сурет. Мысал сөйлемнің грамматикалық талдау ағашы

Ауыстыру сәтті болды деп ереженің сол жағындағы таңбаны оң жағындағы таңбамен ауыстырғанды айтамыз. Сөз қатарын шығару кезінде ол жолда терминал сөздермен бірге терминал емес өрнектер де болуы мүмкін. Бұл суретте келтірілген өзгертулер жоғарыдан- төмен болатын (top-down derivation) трансформациялар деп аталады. Ол sentence (сөйлем) таңбасынан басталып, терминал жолымен аяқталады. Ал төменнен-жоғары болатын трансформация терминалдар қатарынан басталып, ережелердің оң жағындағы элементтерді сол жағындағы элементтермен ауыстырып, sentence таңбасымен аяқталады. Аталған трансформацияны грамматикалық талдау ағашы (parse tree) деп аталған ағаш түрінде бейнелеуге болады. (6.2-суретті қара).

Суреттегі дөңгелекпен – төбелер, төртбұрышпен жапырақ-төбелер деп аталатын граф ұғымдары бейнеленген. Төбелер – грамматика ережелерінің жиынындағы таңбаларды, ал жапырақ-төбелер – терминалдарды, яғни сөздерді білдіреді. Әр төбе және одан тарайтын балалық төбелер грамматикадағы ереженің оң және сол жағын білдіреді. Ағаш түбіріне sententce – сөйлем жатады [9, 10].

Трансформацияның бар болуы немесе талдау ағашының болуы сөйлемнің грамматика тұрғысынан дұрыстығын тексерумен бірге оның құрылымын жасайды. Грамматиканың тіркестік құрылымы (phrase structure) тілдің терең лингвистикалық ұйымдастырылу

сызбанұсқасын анықтайды. Мысалы, сөйлемді етістік және зат есімдік құрылым деп бөлу сөйлемдегі іс-әрекет пен оны тұрғызушы агент арасындағы қарым-қатынас түрін құрады. Мұндай тіркестік құрылым семантикалық жіктеуде негізгі рөл атқарады. Өйткені ол семантикалық өңдеу орындалатын трансформацияның аралық кезеңін анықтайды.

Сөйлемді талдау дегеніміз – трансформация есебін немесе грамматиканы қалыпты түрде анықтау негізінде жасалған енгізу қатарының грамматикалық талдау ағашын құру. Грамматикалық талдау алгоритмдері екі топқа бөлінеді: жоғарыдан-төмен жүретін (*top-down parser*) және төменнен-жоғары жүретін (*down-top parser*) [10]. Бұндай талдауыштардың екі түрі бар:1-ші топтағы талдауыштар өз жұмысын жоғары деңгейдегі *sentence* таңбасынан бастап ағаштың жапырақтары мақсатты сөйлем болатын түрге дейін құрады, 2-ші топтағы талдауыштарда сөйлем сөздерінен басталып рет-ретімен орындалатын операциялар нәтижесінде *sentence* таңбасы қалыптасады.

Грамматикалық талдау есебін шешудің негізгі қиындығы трансформацияның әр қадамында пайдаланылатын ережелерді таңдауда. Бұл ережелер белгілі тәртіппен орналасқан жиындарда болады. Егер таңдау дұрыс орындалмаса, онда талдауыш берілген сөйлемді дұрыс анықтай алмайды. Мысалы, «Бұл ит қабады» деген сөйлемді талдағанда «төменнен-жоғары» әдісін пайдаланайық. Ол үшін 6.1-кестеде келтірілген 7, 9, 11-ережелерді қолдану нәтижесінде *article noun verb* деген қатарды аламыз. Содан кейін қателесу нәтижесінде 2-ші ережені қолдансақ, мынандай жолды аламыз: *article noun-prhase verb.* Бұл жол бізді sentence таңбасына әкеледі. Шын мәнінде, талдауыш 3-ші ере́жені пайдалануы керек. Осындай қиындықтар «жоғарыдан-төмен» әдісін қолданғанда да тууы мүмкін.

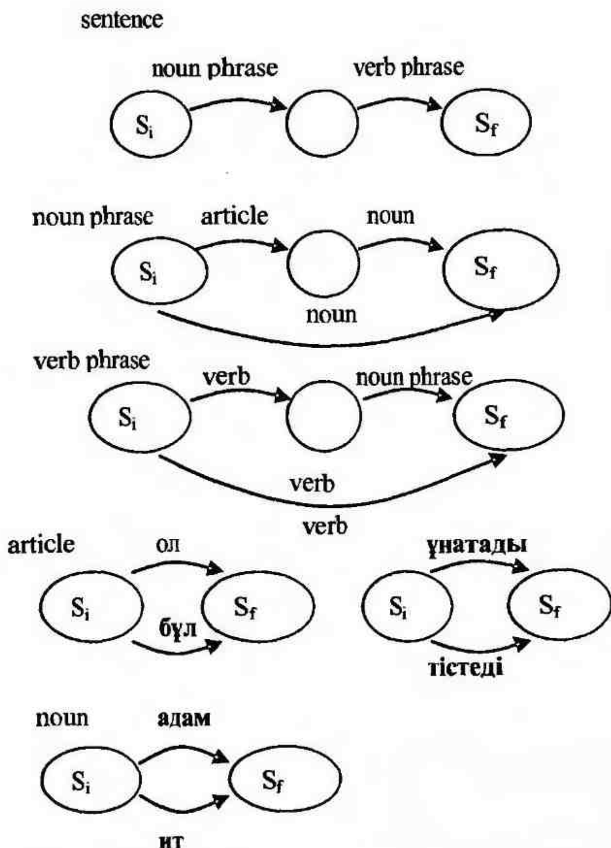
Грамматикалық талдаудың әр кезеңінде дұрыс, қажет ережені таңдау мәселесін шешудің бір жолы қайтару сілтеуіштері деп аталатын көрсеткіштерін орнатып, ереже дұрыс таңдалмағанда бастапқы күйге қайтып келуді ұйымдастыру болып табылады. Осындай шешудің тағы бір тәсілі – ережелерді таңдауға мүмкіндік беретін белгілі бір қасиеттер орнату. Аталған есепке кері есеп ретінде ішкі семантикалық бейнеленуі бар көрсетімнен дұрыс құрылған сөйлемдерді қалыптастыру, яғни генерация есебін шешу.

Генерация сөйлем мазмұнының мағынасын белгілі бір көрсетім арқылы бейнелеп, осы мағынаны беретін грамматикалық дұрыс құрылған сөйлеммен бере алады. Дегенмен генерация есебі оңай шешілетін мәселе емес. Оның шешімін табу үшін, арнайы әдістерді жинақтаған белгілі бір зерттеулер нәтижесі қажет [9].

Грамматикалық талдау әдісін табиғи тіл құрылымдарымен қатар программалық тілдерді талдауда да қолдану тиімді. Сондықтан ғалымдар осындай талдаудың көптеген алгоритмдерін жасаған. Оларда ақпаратты өңдеудің *«төменнен-жоғары»*, *«жоғарыдан-төмен»* стратегияларының екеуі де болады. Осындай алгоритмдердің біріне «өту желілері» деп аталатын алгоритм жатады.

6.3. Өту желілері негізіндегі талдауыштар

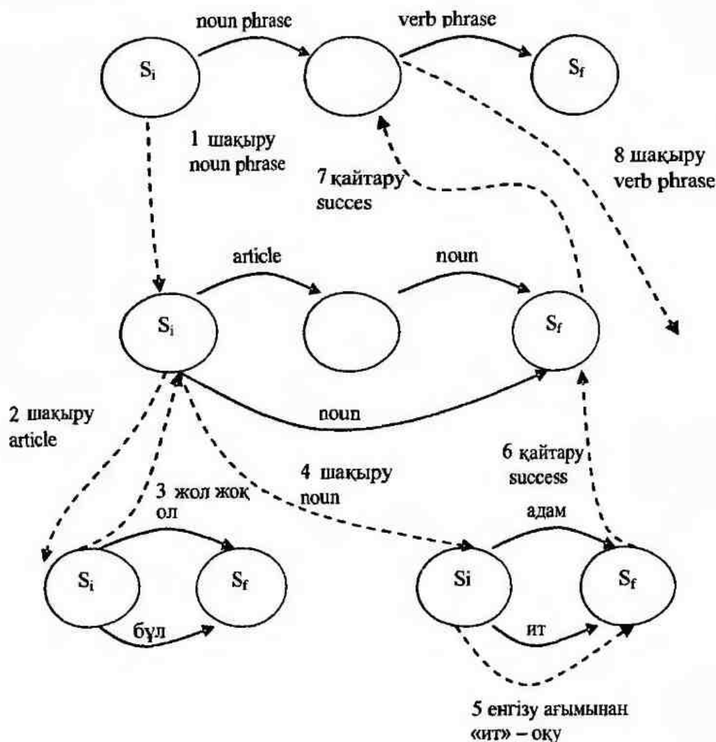
Өту желілерінің негізіндегі талдауыштар өту желілері немесе соңғы автоматтар деп аталатын элементтері бар грамматиканы сипаттайды.



6.3-сурет. Қарапайым грамматикаға арналған өту желісі

Әрбір желі грамматиканың бір терминалды емес элементіне сәйкес келеді [9]. Мұндай желілердегі доғалар терминалдық және терминалдық емес таңбаларымен белгіленеді. Желілердегі бастапқы күйді соңғы күйге келтіретін барлық жолдар терминалды емес элементтерге арналған белгілі бір ережелерден тұрады. Аталған жолдағы, доғалардағы таңбалар тізбектері ереженің оң жағындағы таңбалар тізбектерін береді. Осындай грамматиканың бір түрі 6.3-суретте келтірілген.

Желіде сәтті өтулерді орындау үшін, терминалды емес элементтерді грамматикалық ереженің оң жағындағы элементтермен ауыстыру жеткілікті. Мысалы, сөйлемді талдау үшін, *sentence* сөйлемі желісіндегі өту жолын табу керек. Ол бастапқы күйден (*sintiol*) (6.4-суретті қара, Si) басталып, *noun phrase* өтуін есепке ала отырып, *verb phrase* өтуінен соң соңғы күйге (*Stinol* – суретте – Si) келуі керек. Бұл үдеріс бастапқы *sentence* таңбасын *noun phrase* және *verb phrase* деген таңбалар жұбымен ауыстыруға ұқсайды.



6.4-сурет. Сөйлемді талдау ағашы

Енді доғадан өтуі үшін талдауыш оны тексереді. Егер көрсеткіш-таңбасы терминалды таңба болса, онда талдауыш енгізу ағынын тексереді де, келесі сөз доғаның көрсеткіш-таңбасына сәйкес келетіндігін анықтайды. Егер доға терминалды емес таңбамен белгіленсе, онда талдауыш рекурсивті түрде осы терминалды емес таңба үшін желіге өтеді де, жолды табуға тырысады. Егер жол табылмаса, онда жоғарғы деңгейдегі доғаға жол жоқ. Бұл жағдайда талдауыш бастапқы желіге қайтып келіп, келесі бағыттағы жол түрін тексереді. Осылайша талдауыш sentence желісінде жолдарды табуға әрекет жасайды. Егер қажет жол табылса, онда кіріс жолындағы берілген сөйлем берілген грамматикаға жатады деп табылады.

Мынандай қарапайым сөйлем түрін алайық: «Ит қабады» [9]. Бұл сөйлемді талдауға қажет алғашқы қадамдар төменде келтірілген.

Талдау қадамдары:

1) талдауыш өз жұмысын желідегі *sentence* сөйлемімен бастайды да, *noun phrase* таңбасы бар доғадан өтуге тырысады. Ол үшін ол *noun phrase* желісіне өтеді;

2) *noun phrase* желісінде талдауыш алдымен *article* доғасынан өтуге әрекет жасайды. Ол үшін талдауыш *article* желісіне өтеді;

3) аталған *article* желісінде соңғы төбеге (күйге) өтетін жол жоқ, өйткені бірде-бір доғада сөйлемдегі «ит» деген сөзбен белгіленген көрсеткіш-таңба жоқ. Осындай сәтсіз әрекеттерден соң, талдауыш *noun phrase* желісіне қайтып келеді;

4) енді талдауыш желідегі *noun phrase* желісінің noun доғасына өтуге тырысады да, *noun* желісіне көшеді;

5) «Ит» сөзінен белгіленген таңба доғасынан өту сәтті болады, ол таңба енгізу ағынындағы сөйлем қатарының бірінші сөзіне сәйкес келеді;

6) сонымен, *noun* желісіндегі іздеу оң нәтиже береді. Ол – *noun* таңбалы доғасы бар *noun phrase* желісі соңғы мақсатты күйге апарады деген сөз;

7) көрсетілген *noun phrase* желісі жалпы желінің ең жоғарғы деңгейіне оң нәтиже беріп, *noun phrase* доғасына өтуге рұқсат етеді. Жоғарыда аталған қадамдарға ұқсас.

Әрекеттер сөйлемнің *verb phrase* бөлігінде де осылайша қайталанады. Төмендегі келтірілген өту желілеріне негізделген талдауыш қызмет көрсететін псевдокод берілген [9]. Мұнда өзара рекурсивті болатын екі түрлі функция жұмысы көрсетілген. *Parse* және *transition* деген функциялар. *Parse* функциясының аргументі – грамматика

таңбалары. Егер ол терминалды таңба болса, онда *parse* функциясы оны енгізу ағынының келесі сөзімен салыстырады. Егер терминалды болмаса, *parse* функциясы осы таңбаға байланысты өту желісіне барады да, осы желідегі жолды табу үшін, *transition* функциясын шақырады. Аталған функция параметр есебінде өту желісінің күйін пайдаланып, осы желідегі жолды тереңнен іздеу әдісі көмегімен табуға әрекет жасайды. Сөйлемді талдау үшін, *parse (sentence)* функциясы шақырылады.

```
Function parse (грамматика таңбасы);
Begin енгізу ағымындағы сілтеуішті ағымдағы орынға сақтау;
   case грамматика таңбасы – терминал:
 if грамматика таңбасы енгізу ағымындағы келесі сөзге сәйкес келсе;
         then return (success);
         else begin;
             return (failure) енгізу ағымын қайтадан орнату;
         end; грамматика таңбасы терминалды емес;
    begin грамматика таңбасына сәйкес келетін өту желісіне бару;
         күй:=желінің бастапқы күйі;
         if transition (күй) қайтарады;
         success;
         then return (success);
         else begin;
             енгізу ағымын қайтадан орнату;
             return (failure);
         end;
   end;
 end;
end.

Function transition (ағымдағы күй);
Begin;
   Case ағымдағы күй – соңғы күй:
      Return (success) ағымдағы күй – соңғы күй емес:
         While ағымдағы күйде тексерілмеген өтулер бар;
             Do begin грамматика таңбасы:= келесі тексерілмеген
             өту таңбасы;
                If parse (грамматика таңбасы) қайтарады success;
                   Then begin келесі күй:= өту соңындағы күй;
                      If transition (келесі күй) қайтарады;
                         Success;
```

```
                        Then return (success);
                    End;
                End;
            Return (failure);
        End;
End.
```

Егер талдауыш қате жіберіп, бастапқы күйге қайтып келуге мәжбүр болса, онда *parse* функциясы енгізу ағымындағы сілтеме көрсеткішті ағымдағы орнында сақтайды. Олар егер талдауыш қайта айналып келгенде енгізу ағынын қайтадан орнатуға мүмкіндік береді.

Мұндай өту желілеріне негізделген талдауыш сөйлем дұрыстығын анықтайды, бірақ ол грамматикалық талдау ағашын құра алмайды. Бұл мақсатты орындау үшін, талдау ағашының бір бұтағы ретінде жүретін қайтару функциясын құру қажет. Ол үшін талдау процедурасын былайша өзгертуге болады:

1) онда *parse* функциясының параметрі есебінде терминалды таңба жүреді. Ол таңба енгізу ағынының келесі таңбасына сәйкес келіп отыруы керек. Әрдайым *parse* функциясын шақырған сайын ол осы таңбамен белгіленген бір жапырақ түйіннен тұратын ағаш түрін қайтару қажет;

2) егер *parse* функциясы грамматика таңбасының терминалды емес параметрі үшін шақырылса, онда ол *transition* функциясын шақырады. Аталған функция қызметі сәтті аяқталса, онда ол ағаш бұтақтарының реттелген жиынын қайтарады. *Parse* функциясы осылардың негізінде түбірі – грамматика таңбасы болатын, ал балалық элементтері *transition* функциясын қайтаратын бұтақтар болатын ағашты құрады;

3) желідегі қажет жолды іздегенде *transition* функциясы шақырылады. Бұл функция қызметі сәтті аяқталса, талданған таңба нәтижесін беретін ағаш түрі табылады. *Transition* функциясы бұл бұтақтарды реттелген жиын есебінде сақтайды да, желідегі жолдарды тапқанда, осы жол доғаларындағы көрсетілген таңбаларға сәйкес талдау ағашының реттелген жиынын қайтарады.

6.4. Хомский иерархиясы

Жоғарыда талданған сөйлемдер мысалы *мәнмәтіндік* тәуелсіз грамматика (*context-free grammor*) негізінде құрылады. Мұндай грамматикада ереженің сол жағында бір терминалды емес таңба болады. Демек, ережені сөйлемнің ***мәнмәтіндік*** мағынасына және тәуелсіз мәтінің кез келген таңбасына қолдануға болады [9].

Компьютерлік ғылымда программалау тілдерін анықтау үшін аталған ережелер өте қуатты құрал болғанымен, олар табиғи тіл ережелерін құруда аса тиімді бола бермейді. Атап айтсақ, жоғарыда келтірілген мысал сөйлемдердегі зат есім, етістік түрлерін берсек, онда оның грамматикасы мынандай болады:

noun<-> адамдар;
noun<-> иттер;
verb<-> қабады;
verb<-> ұнатады.

Бұл ережелер көмегімен мына сөйлемдердің грамматикалық талдауын іске асыруға болады: «Бұндай иттер осындай адамдарды ұнатады». «Осы адамдар мұндай иттерді ұнатады». Мұндағы одағай сөздер мен зат есімдер арасындағы байланыстар «бұлыңғыр» түрде берілген. Талдауыш мұндай сөйлем түрлерін қабылдап алғанымен, сөйлем мүшелерінің көпше және жекеше түрлеріне талдау жасай алмайды. Мысалы, сөйлемді анықтайтын *sentence* ережесі заттық құрылым (*noun_phrase*) соңынан жүретін етістік құрылымындағы (*verb_phrase*) көпше түрінің жалғауларының сәйкестігін тексермейді («Адам+дар, Ит+тер»). Ондай сәйкестіктің одағай сөздер мен сын есімге де қатысы бар («Кәрі Адам+дар») («Кішкене Ит+тер»).

Осы *мәнмәтіндік* тәуелсіз грамматика түрінде анықталған тілдер басқа грамматикалармен бір қалыптағы тілдер иерархиясының тобына кіреді. Осындай иерархия *Хомский иерархиясы [chomsky hierarchy]* деп аталады. Мұндай иерархияның ең төменгі деңгейінде *регулярлы тілдер [regular language]* деп аталатын класс жатады. *Регулярлы тіл* деп грамматикасы ақырғы автоматтарды пайдаланып, анықталған тіл түрін айтамыз. Бұндай тілдер компьютер ғылымдарында жиі қолданыста болғанымен, олар көптеген программалау тілдеріндегі синтаксисті көрсетуге аса тиімді емес. *Мәнмәтіндік* тәуелсіз грамматика тілдері (*context-free language*) Хомский иерархиясында *регулярлы тілдер* деңгейінен жоғары орналасады. Олар белгілі бір тәртіппен берілген шығару ережелері көмегімен анықталады. *Мәнмәтіндік* тәуелсіз грамматика ережелердің сол жағында тек бір терминалды емес таңба болады. *Мәнмәтіндік* тәуелсіз грамматика тілдер тобын «өту желілері» алгоритмдері көмегімен талдауға болады. Егер «өту желілері» негізіндегі талдауышта рекурсияға тыйым салынса (яғни доғаларды тек терминалды таңбалармен белгілесе, олар басқа желілерді

«шақырмайды»), онда мұндай тілдер тобы *регулярлы өрнектер* жиынына жатады. Сонымен, *регулярды тілдер – мәнмәтіндік* тәуелсіз грамматика тілдердің жиынына кіретін ішкі жиын.

Мәнмәтінді тәуелсіз тілдердің тағы бір тобына рекурсивті-тізімделген тіл (*recursiveey enumerable language*) жатады. Мұндай тілдер шектелмеген продукциялық ережелер көмегімен анықталады. Бұл тілдің компьютерлер ғылымы үшін маңызы болғанымен, табиғи тілдің синтаксисін толық анықтау мүмкіндігі аз. *Мәнмәтінді* тәуелсіз грамматиканың қарапайым түрі (артикль – зат есім – етістік) (*article – noun verb*) келесі 6.5-суретте келтірілген [9].

Бұл суреттегі аталған ережелерде артикль мен зат есім, зат есім мен етістік сияқты сөйлем мүшелерінің көпше және жекеше түрлерінің өзара келісілген түрлері берілген. Шектеулер әртүрлі артикльдердің, зат есімдердің, етістіктердің көпше және жекеше түрлерінің өзара сәйкес келуін қамтамасыз ететін ережелерді анықтауда пайдаланылады.

```
sentence<->noun_ phrase verb_ phrase,
noun_ phrase<->article number noun,
noun_ phrase<->number noun,
number<->singular,
number<->plural,
article singular<->a singular,
article singular<->the singular,
article plural<->some plural,
article plural<->the plural,
singular noun<->man singular,
singular noun<->dog singular,
plural noun<->men plural,
plural noun<->dogs plural,
singular verb_ phrase<->singular verb,
plural verb_ phrase<->plural verb,
singular verb<->likes,
singular verb<->bites,
plural verb<->like,
plural verb<->bite.
```

6.5-сурет. Мән мәтінді тәуелсіз грамматиканың қарапайым түрі

Мысалы, «Бұл иттер қабады» деген сөйлемнің жоғарыда келтірген грамматика бойынша жіктелуі былайша орындалады.

sentence.
noun_ phrase verb_ phrase.
article plural noun verb_ phrase.
Бұл plural noun verb_ phrase.
Бұл иттер plural verb_ phrase.
Бұл иттер plural verb.
Бұл иттер қабады.

Жоғарыдағы 6.5-суретте шектеулердің әр түрлері көрсетілген. Бұл грамматикада қолданылған *singular, plural* – терминалды емес таңбалары – шектеуді қамтамасыз етеді.

6.5. Табиғи тілді түсіну жолдары

Мәнмәтінді тәуелді тілдер. Мәнмәтінді тәуелді тілдер (*context-sensitive-language*) мән-мәтінді тәуелді грамматика негізінде анықталады. Олар ереженің сол жағындағы бірнеше таңбаны пайдаланады. Яғни мәнмәтінді анықтау үшін қандай ереже іске қосылатынын табады. Осылайша ауқымды түрдегі шектеулер анықталады, яғни сөйлем мүшелерінің көпше және жекеше түрлерінің келісімді түрге келуі тексеріледі. Мәнмәтінді тәуелді грамматика ережелері үшін қойылатын негізгі шектеудің тағы бір түрі – ереженің оң жағы мен сол жағындағы таңбалар саны бірдей болуы керектігінің талабы. Мәнмәтінді тәуелді грамматиканы синтаксистік келісімдерді орындауды тексеру үшін де пайдалануға болады. Мысалы, осы грамматикаға терминалды емес мынандай «act_of_bitng» таңбаны қосу арқылы «Адам итті қабады» деген сөйлемнің қолданылуына тыйым салуға болады. Бұл терминалды емес таңба «адам» деген зат есімнің «қабады» деген етістікпен қолданылатын сөйлем түрін құруға кедергі жасайды. Мәнмәтінді тәуелді грамматикалар мәнмәтінді тәуелсіз грамматикалар ала алмаған тілдік құрылымдарды анықтауға көмегін тигізеді. Дегенмен оларды іс жүзінде қолданатын талдауыштар құрудың бірталай қиындықтары бар [9]. Ондай қиындықтарға мыналарды жатқызамыз:

1) Мәнмәтінді тәуелді грамматиканы пайдаланғанда, терминалды емес таңбалар мен ережелер саны күрт өседі. Көз алдыңызға сөйлем мүшелерінің көпше және жекеше түрлерін, жақтарын

(1-ші, 2-ші) бір-бірімен байланысын сипаттайтын мәнмәтінді тәуелді грамматика түрін елестетіп көріңіз.

2) Мәнмәтінді тәуелді грамматикаларда тілдегі сөз тіркестерінің құрылымдарын анық көрсету мүмкіндігі аз.
3) Тілдің синтаксистік және семантикалық құрамдас бөлшектерін бөлшектегенде, грамматиканың өзінің семантикалық бірыңғайлылығы бұзылуы мүмкін.
4) Мәнмәтінді тәуелді грамматикалар мәтіннің мағынасын беретін семантикалық құрылымдар құру мәселесін шеше алмайды.

Талдауға түскен сөйлемді қабылдайтын немесе мүлдем кері қайтару қызметін ғана атқаратын талдауыш ешкімге қажет емес. Ол сөйлемнің мағынасын қайтаруы керек. Сонымен, мәнмәтінді тәуелді грамматика басқа грамматика түрлеріне қарағанда өз ерекшеліктері бар табиғи тілді пайдалану мүмкіндіктерін беретін құрылымдарға жатады.

Қорыта айтсақ, мәнмәтінді тәуелді грамматика басқа грамматика түрлеріне қарағанда, табиғи тілді пайдалануда көптеген ерекше мүмкіндіктерді құратын құрылымдарға жатады. Бұл грамматикада ауқымды түрдегі шектеулер анықталады, яғни сөйлем мүшелерінің көпше және жекеше түрлерінің келісімді түрге келуі тексеріледі. Мәнмәтінді тәуелді грамматика ережелері үшін қойылатын шектеулер сөйлемнің мағынасын қайтаруда ерекше рөл атқарады.

6.6. Табиғи тілдегі сөз қорын құру

Сарапшы жүйе түріндегі компьютер программалары пайдаланушы адаммен табиғи тілде сұхбат жүргізеді. Жасанды зерде теориясының зерттеулерінде табиғи тіл мәселесіне көп орын беріледі. Қазіргі кезде белгілі тақырыпқа қарапайым сөйлемдер құрылысын енгізу арқылы сұхбат жүргізе алатын компьютерлік программалар жасалған. СЖ программаларындағы пайдаланушы мен сұхбат жүргізетін программалар сипаты мынадай болып келеді: кез келген түрде берілген қарапайым сөйлемдерден тұратын жауаптарды түсініп, қорыту қабілеті; білім қорындағы берілген жағдайларға сәйкес, программаның сұрақтар қоя білу қабілеті; қажет болғанда өзі келген қорытындысын табиғи тілде түсіндіре алу қабілеті.

Осындай мәселенің бір мысалы ретінде кез келген салаға қатысты белгілі бір сөз қорын құру алгоритмін қарастырайық. Сөз қорында сала аумағындағы мамандарға таныс терминдерден тұратын сөздер бар дейік. Мысалы, «Ақпаратты қорғау» аумағына қатысты бірнеше сөзден

тұратын деректер қоры бар делік. Дерекқор құрылымы реляциялық түрде жобаланған болсын. Яғни аталған тақырыптағы сөздер келесі көрсетілген кестелерде болады.

6.2-кесте

Рет	Өріс аты	Сипаты
1	Сөз аты	Таңба
2	Сөз анықтамасы	Таңба

6.2-кестеде дерекқордың құрылымдық түрі келтірілген. Яғни кестеде екі өріс түрі бар және жазбалар сипаты таңбалардан тұрады.

Екінші кестеде (6.3-кестені қара) дерекқордағы болатын жазбалар келтірілген. Мысал үшін 12 жазбадан тұратын дерекқор түрін алдық.

6.3-кесте

Рет	Сөз аты	Анықтамасы
1	2	3
1	Ассиметриялық жүйе	Ашық кілтті криптожүйелер деп те аталады. Бұл жүйеде деректерді шифрлау үшін бір кілт, шифрды ашу үшін басқа кілт қолданылады
2	Гаммалау (gamma xoring)	белгілі бір заңға сәйкес шифрдің гаммасын ашық деректердің үстіне салу (беттестіру) үдерісі.
3	Криптология	Ақпаратты оны түрлендіру арқылы қорғаумен шұғылданатын ғылым
4	Криптография	Құпия жазу, ақпаратты заңсыз пайдаланушылардан қорғау мақсатымен оны түрлендіру әдістері жайындағы ғылым
5	Криптоберіктілік	Шифрдің негізгі сипаттамасы. Ол кілт белгісіз жағдайда шифрдің кері шифрлеуге беріктігін анықтайды
6	Цифрлық Су Таңбалар (ЦСТ) (Цифровой водяной знак – watermarking)	ЦСТ өз атауын бағалы қағаздарды, оның ішінде ақшаларды қолдан жасаудан арашалап алу сияқты қорғау тәсілінен алынған

6.3-кесте жалғасы

1	2	3
7	Стегожүйелер	Стеганография ғылымының әдістерін пайдаланып құрылатын жүйелер Стегожүйенің құрамында міндетті түрде адам баласы болуы керек. Яғни адам деректерді қабылдап алатын қосымша қабылдағыш ретінде қарастырылады
8	Прекодер	Жіберілетін хабарды сақтайтын ақпараттық тізбектерден тұратын контейнер, жасырылған хабарды деректер арасына енгізуді іске асыруға арналған құрылғы, енгізілген хабарды ерекшелеуге арналған құрылғы
9	Стегодетектор	Стего хабардың бар екенін анықтайтын стегодетектор, жасырын хабарды қалпына келтіруге арналған декодер, жасырын хабардың құпиялығын арттыратын кілт, анықталған ЦСТ-ты бөліп алуға арналған стегодекодерлер болады
10	Робасталық	Цифрлық су таңбаларының стегожүйеге әр түрдегі әсер еткен кездегі болатын тұрақтылығы
11	Стегожол	Санауларда өзгерістері бар биттер тізбегі. Хабарды контейнерге енгізу бір немесе бірнеше кілт көмегімен іске асады. Кілт дегеніміз – генератор арқылы туындайтын биттердің псевдокездейсоқ тізбегі. Жасырылатын ақпарат белгілі бір кілт бойынша санауларға енгізіледі
12	Сынғақты (хрупкие)	Цифрлық су таңбалары толтырылған контейнерге аздап болса да өзгерістер жасағанда бұзылады

Осылайша анықталып құрылған сөз қорын толықтыру үшін, келесі қадамдардан тұратын алгоритмді ұсынуға болады. Бұл алгоритмдегі табиғи тілді түсіну осы салаға қатысты құрылған сарапшы жүйенің білім қорындағы білімдерді толықтыру үдерісін бақылау арқылы ғана іске асу мүмкіндігін береді. Білім құрылымы < «Сөз аты», «Сөз анықтамасы» > деп көрсетілген екілік түрінде берілген. Дерекқордағы бұл екі өріс арасындағы релятивтік байланыс түрі ғана пайдаланылған. Яғни жүйенің өз білімін толықтырып, «түсіне» білу қабілеті «Сөз анықтамасы» өрісіндегі сөздерді өзара салыстырып тексеру механизмімен ғана шектеліп тұр. Егер бұл өрістегі жазбаға жоғарыдағы шолулардағы келтірілген грамматикалық талдау немесе мәнмәтіндік тәуелді, тәуелсіз грамматикалар механизмдері қолданылса, онда компьютерлік бағдарламаның білімді адам сияқты «қорытып» талдай білу қабілетін иеленуге жақындай түсуі әбден мүмкін.

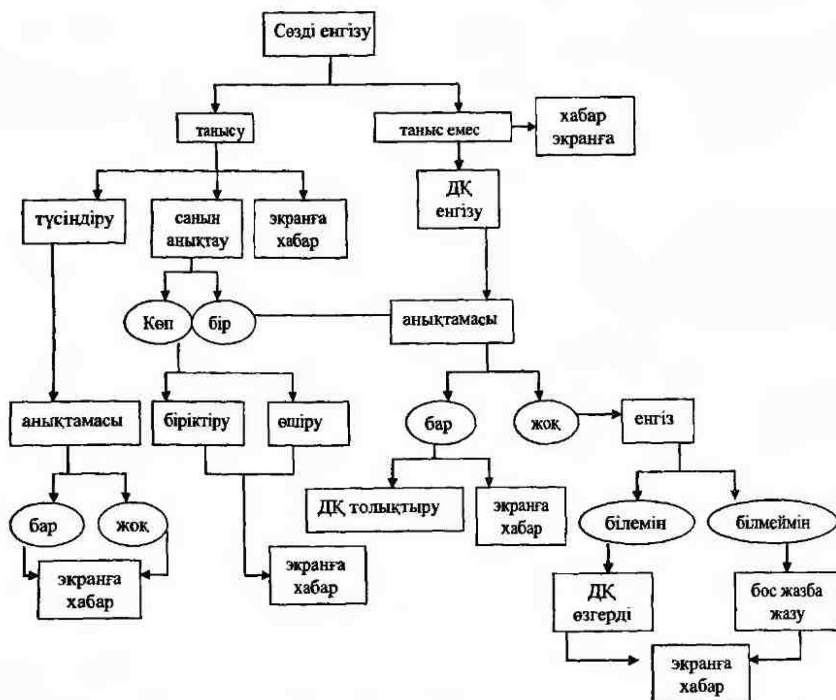
Сөз қорын толықтыру алгоритм қадамдары

1. Енгізілетін сөз уақытша дерекқорға жазылады.
2. Енгізілетін сөз – сөз қорындағы бар сөздермен оның «Сөз аты» деген өрісіндегі жазбамен салыстырылады. Егер ондай сөз болса, онда басқару төртінші қадамға беріледі, керісінше жағдайда келесі қадамға беріледі.
3. Егер ондай сөз болмаса, ол сөз қорына жазылып, пайдаланушы үшін экранға қосымша хабар шығады.
4. «Сөз анықтамасы» деп аталатын өрісте жазба бар болса, енгізілген сөз сонымен салыстырылады, егер анықтама болса, басқару жетінші қадамға беріледі, керісінше жағдайда келесі қадамға беріледі.
5. Егер аталған өрісте жазба болмаса, пайдаланушыға оны енгізу жөнінде нұсқау беріледі.
6. Пайдаланушы енгізген сөз анықтамасы «Сөз анықтамасы» деп аталатын өрістегі жазбаға жазылады.
7. Енгізілген сөздің дерекқорда неше рет кездесетіні тексеріледі. Егер сөз анықтамалары бірдей болмаса, онда екі жазба бір жазбаға біріктіріледі (мысалы, «Криптоберіктілік» сөзі. Оның екі түрлі анықтамасы болса), керісінше жағдайда келесі қадамға беріледі.
8. Егер «Сөз аты» және «Сөз анықтамасы» деп аталатын өрістердегі жазбалар бірдей болса, бір жазба өшіріледі.
9. Егер пайдаланушы сөз анықтамасын білмесе, онда жаңа сөз – сөз

қорының «Сөз аты» деп аталатын өрісіндегі жазбаға жазылады да, «Сөз анықтамасы» деп аталатын өрістегі жазба бос қалып, экранға ол туралы қосымша хабар шығады.

10. Енгізілетін жаңа сөздер мен олардың анықтамалары енгізілген соң, сөздер сөз қорына жазылады.
11. Егер пайдаланушы сөз қорындағы бар сөздер туралы немесе жаңадан енгізілген сөздер туралы мағлұмат білгісі келсе, онда «Түсіндіру» деп аталатын блок көмегімен қосымша кеңесті алуға болады.

Бойына осындай қадамдарды жинақтаған алгоритм қадамдарының сызбанұсқалық түрі 6.6-суретінде келтірілген.



6.6-сурет. Сөз қорын толықтыру алгоритмі сызбанұсқасы

Жүйенің «таныс» сөзді тани білу қабілеті бұл алгоритмде қарапайым түрдегі салыстыру нәтижесінде санмен анықталады. Яғни оны алгоритмдік тілдерде кездесетін белгілі бір тәсілдермен де

басқаруға болады. Бұл алгоритмнің тағы бір ерекшелігі – компьютерлік программаның пайдаланушымен сұхбат түрінде үнемі қарым-қатынаста болуы. Яғни «Экранға хабар» деп аталатын модуль арқылы келесі қадамды пайдаланушы өзі анықтап тұрады. Бұл көрсеткіш те сарапшы жүйелерге тән компьютердің «адамша» сөйлесу қабілетін одан ары қарай дамыта алу мүмкіндігі бар екенін көрсетеді. Дегенмен бұл қарапайым мысалда салынған компьютер мен пайдаланушы арасындағы «сөйлесу» механизмі де қарапайым. Ол көбінде «жоқ» немесе «иә» деп аталатын сұрақ – жауап төңірегінде дамиды. Оған «адамдық» қасиеттерді беру үшін, білім қорына қатысты теориялық зерттеулерге көңіл аударып, білім бірлігі болып саналатын «Фрейм» немесе «Семантикалық тор» құрылымдарын пайдаланып, сала маманы қолданатын терминдердің түсіндірме қорын жасау қажет. Бұл қорды бөлек түрде де, немесе білім қорының құрылымын жобалағанда бірден құруға да болар еді.

Жоғарыда келтірілген алгоритм қадамдарын белгілі бір процедураларды пайдаланып, алгоритмдік тілдерде де, нысан бағытталған тілдерде де өзіміз анықтаған бір салаға қатысты сөз қорын құрып, оны өзгертіп, толықтырып тұруға болады. Егер келтірілген алгоритм түрін логикалық немесе функционалдық программалау стилінде беретін болсақ, онда оған деген талап-тілектердің сәл өзгеше болатынын атап өтуге болады. Мысалы, Пролог тілінде жоғарыда құрылған дерекқорда міндетті түрде предикаттар, яғни «шын» мәнін қабылдайтын фактілер болады. Мысалы, 6.2-кестедегі дерекқор жазбасының кейбір түрлері Пролог тілінде былайша өрнектеледі:

Аталады (ассиметриялық жүйе, ашық кілтті криптожүйелер).
Аталады (гаммалау, гамманы деректердің үстіне салу үдерісі).
Аталады (криптология, ақпаратты қорғаумен шұғылданатын ғылым).
Аталады (криптоберіктілік, кері шифрлауға беріктігі).
Аталады (цифрлық су таңбалар, ақшаны арашалау тәсілі).
Аталады (стегожүйелер, адам деректерді қабылдағыш).
Аталады (прекодер, хабарды сактайтын контейнер).
Аталады (стегодетектор, хабардың құпиялығын арттыратын кілт).
Аталады (робасталық, ЦСТ тұрақтылығы).
Аталады (стегожол, санауларда өзгерістері бар биттер тізбегі).
Аталады (сынғақты, ЦСТ өзгерістерінің бұзылуы).

Байқап отырғаныңыздай, реляциялық дерекқордағы өріс жазбалары мен Пролог дерекқорындағы аргументтер арасында біраз

айырмашылықтар бар. Ол екі қорда да салыстыру механизмі бірдей болғандықтан келтіріліп отыр. Егер, біз жоғарыда атап өткендей, «Сөз анықтамасы» өрісіндегі сөйлемдерге жоғарыдағы шолулардағы келтірілген грамматикалық талдау немесе мәнмәтіндік тәуелді, тәуелсіз грамматикалар механизмдері қолданылса, онда, әрине, Пролог тіліндегі фактілерді басқаша берген болар едік. Ондай жағдайда предикат «аталады» етістігімен ғана шектелмей, сөйлемдегі басқа да етістік түрлерін қамтиды. Бұл аталған мәселені келесі зерттеулерімізде талқылаймыз ба деген үміт бар.

Бақылау сұрақтары:

1. Табиғи тілде сұхбат жүргізу әдістері.
2. Табиғи тілді талдаудың әртүрлі деңгейлері қандай?
3. SHRDLU жүйесінің ерекшеліктері.
4. Таңбалы талдау деген не?
5. Синтаксистік талдау ерекшеліктері.
6. Грамматикалық талдау алгоритмдері.
7. Өту желілерінің негізіндегі талдауыштар деген не?
8. Сөйлемді талдау ағашы неге қажет?
9. Хомский иерархиясы деген не?
10. Мәнмәтіндік грамматика деген не?
11. Тәуелсіз грамматика (context-free grammor) деген не?
12. Табиғи тілде сөз құрын кесте арқылы құру әдісі.
13. Табиғи тілде сөз құрын Пролог стилінде құру әдісі.

ҚОРЫТЫНДЫ

Қоғамдық өнеркәсіптің ары қарай дамуы ғылыми-техникалық прогрестің жетістіктеріне негізделген. Ол осы алдыңғы қатардағы ой жетістіктерін өндірістік тәжірибеде қолдану, жаңа техника мен технологияларды тиімді пайдалану мүмкіндігіне байланысты болады. Қазіргі таңда осы технологиялар ішіндегі ақпараттық технологиялар ең озық, ең алдыңғы қатарда тұр деуге болады.

Бұл оқу құралының көздеген негізгі мақсатарының бірі – жасанды зерде есептеріне арналған программалау тілдерін таңдауға байланысты негізгі мәселелерді қарастыру және зерделік есептерді шешуге қажетті басқару мен деректерді көрсету құрылымдарын қалыптастыру болып табылады. Осындай мақсатты іске асыру тілдеріне қажетті қасиеттерді, негізінен, осы құрылымдарға қойылатын талаптар анықтайды. Зерделік жүйелерді сипаттау екі түрлі: білімдер және таңбалар деңгейінде өтеді. Жасанды зерде тілдерінде программалау әдістері функционалдық және логикалық программалау тілдерімен тығыз байланысты. Бұл программалаудың мағлұмдамалық стиліне жатады және Лисп, Хаскелл және Пролог тілдері – осындай стиль тілдері. Бұл тілдерге қойылатын талаптар: тілдер құрылымы компьютерлік архитектураның төменгі деңгейімен, операциялық жүйелер және аппараттық құралдар архитектурасы деңгейімен, жады көлемі, процессор жеделдігімен шектеулі болуы қажет. Сонымен бірге бұл тілдер жаңа заманғы программалау тілдеріне енетін жаңа құралдар негізін құруда көп рөл атқарады. Пролог тілі жоғарғы деңгей тіліне жатады. Оның қатаң түрде жүйеленген теориялық негіздемесі бар. Ол математикалық логиканың тұжырымдамасы мен әдістерін пайдалануға бағытталған. Прологтың басқа тілдерден ерекшелігі – онда жазылған программалардың мағлұмдамалық сипатта болуы.

Қазіргі таңда программалық қамтама құрамы күн өткен сайын күрделене түсуде. Әдеттегі программалау тілдерінде модульдік құрылымды программалар құруға тұжырымдық шектеулер бар. Функционалды тілдер бұл шектеулерді алып тастауға мүмкіндік береді. Программаның модульдік қасиетін арттыру үшін, функционалдық тілдерде екі түрлі ерекшелік қарастырылады. Ол ерекшелікке жоғары дәрежелі функция және «жалқау есептеулер» деп аталатын есептеу түрін пайдалануды жатқызамыз. Жалпы, функционалдық атаудың өзінде программалаудың осы түрінің негізі жатыр. Өйткені программаның өзі функциядан тұрады. Программа өзіне қажетті

бастапқы деректерді функцияның аргументі есебінде қабылдап алып, нәтижені функция нәтижесі есебінде береді. Әдетте, негізгі функция басқа функциялар терминдері аумағында анықталады, ал олар одан ары қарай тағы да басқа функция аумағында анықталып, осылайша жалғаса береді. Ең соңындағы функция ең төменгі деңгейдегі тіл бірлігі болады.

Оқу құралында жасанды зерде есептерінің өздеріне қатысты біраз әдістерге шолу жасалған. Жасанды зерде деп саналы тәртіпті басқаруға автоматтандыратын компьютерлік ғылымның аумағын атаймыз. Жалпы, «Зерде – Интеллект» деген не? Яғни оны қанша мөлшерде құруға болады және ол қандай мөлшерде алдын ала табиғатта бар? «Шығармашылық» деген не, түйсіну деген не, интеллект бар екенін байқап, қадағалап көреміз бе, әлде оны басқаратын жасырын күш бар ма? Тірі заттардың нерв клеткаларындағы білімдер қалай құрылған және зерделік құрылғыларды жобалағанда оны қалай қолдануға болады? Өзіндік талдау деген не және ол шамалықпен қалай байланысты болады, зерделік компьютерлік программаларды адам санасына ұқсас құру керек пе, әлде қатаң түрде «Инженерлік» жол жеткілікті ме? Саналылықты компьютер техникасы көмегімен құруға бола ма? Әлде зерде мағынасы тек биологиялық жәндіктерге тән сезім мен тәжірибе арқылы ғана анықтала ма? Осындай көптеген сұрақтар төңірегіндегі зерттеулер адам баласын көптен бері толғандыруда. Бұл сұрақтардың барлығына толық жауап әлі жоқ, дегенмен осы сұрақтар төңірегінде өрбіген мәселелер мен оның әдістемелік жолдарын зерттеу қазіргі заманғы жасанды зерде ғылымы саласының негізін құрды.

Ақпаратты өңдеу саласында осы уақытқа дейінгі қолданып келген әдіс, тәсілдер математикалық дәл зерттеулерге негізделіп келді. Қазіргі кезде адамға ғана тән пайымдаулар мен тұжырымдарды жасайтын зерттеулер жасанды зерде бағытында қарастырылады. Осының бір саласы ретінде сарапшы жүйе деп аталатын зерттеудің ғылымда алатын орны ерекше. Сарапшы жүйе тәсілдері негізінде құрылған білімдер мен деректер құрылымдарын көптеген салада қолдануға болады. Осындай әдістермен құрылған компьютерлік программалардың көпшілігі қазірдің өзінде коммерциялық өнім есебінде пайдаланылып жүр. Болашақта жасанды зерде бағытының маңызды саласының бірі болып саналатын сарапшы жүйе зерттеулерінің нәтижесі адамның ұшқыр қиялын одан әрі алға жетелеп, адамға тән қасиеттердің кейбірін компьютер жадысында үлгілеуге мүмкіндік береді.

Сарапшы жүйе түріндегі компьютер программалары пайдаланушы адаммен табиғи тілде сұхбат жүргізеді. Оның сипатына: кез келген түрде берілген қарапайым сөлемдерде тұратын жауаптарды түсініп, қорыту қабілетін, білім қорындағы берілген жағдайларға сәйкес, программаның сұрақтар қоя білу қабілетін, қажет болғанда өзі келген қорытындысын табиғи тілде түсіндіре алу қабілетін жатқызамыз. Тілді түсіну деген ондағы сөздер мен сөйлемдерді айту ғана емес. Ол үшін аталған тілде сөйлеп тұрған адам сала мәселесінің түсінігін де сол тілде жеткізе білуі керек.

Зерделік жүйелердегі табиғи тілді түсіну жолдары мәселе шешілетін аумақтағы білімдерге тікелей байланысты болады. Табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдердің анықталған пайдалану құрылымдарын қажет етеді. Осындай күрделі үдерістерді басқару үшін тіл мамандары табиғи тілді таңдаудың әртүрлі деңгейлерін анықтады. Осындай деңгейлерге: Просодия, Фонология, Морфология, Синтаксистік талдау, Семантика, Прагматика, Төңіректегі дүние жайлы білім деп аталатын жеті түрін жатқызуға болады. Осындай құрылым түрлерін жасағанда білімдердің өзгешелігін білдіретін қасиеттерімен бірге, адамдар арасында болатын күрделі қарым-қатынастар түрлерін, бір сөздің бірнеше мағынада айтылуы, үйрену әдістері, әртүрлі көзқарастар сияқты мәселелердің барлығын есепке алу қажет. Оқу құралында осындай деңгейлерді зерттеу мәселесімен бірге, сарапшы жүйе интерфейсіне қатысты сұхбат жүргізу элементтеріне де талдау жасалған.

Кітапта табиғи тілді түсіну мәселесіне қатысты сарапшы жүйе, зерделік жүйе әдістері мен құралдарына зерттеу жасалған. Тілді түсінуге қатысты білімді пайдалану құрылымдары, табиғи тілді талдау кезеңдері, құрылған тілдің талдауыштар түрлері, мәнмәтінді тәуелді, тәуелсіз тілдер, грамматикалар деген сияқты құрылымдарға анықтамалар келтіріліп, олардың әр түрлері талданған. Табиғи тілді түсінудегі тіл құрылымдарының деңгейлеріне шолу жасалып, оның интерфейс құралдары зерттелген. Табиғи тілді түсінуді іске асыратын компьютер программалары тілдегі жинақталған білімдерді пайдаланудың құрылымдарын қажет етеді. Ол үшін табиғи тілдегі деңгейлер санын анықтау қажеттігі туады. Жұмыста осындай талдаулар жүргізілген. Белгілі бір сала мәселесіне қатысты сөз қорын құруға қатысты алгоритм түрі құрылған.

Сонымен бірге оқу құралында үйрету жүйелері жайында да біраз мағлұмат берілген. Компьютер жадында білім формасындағы

ақпаратты жинақтап сақтау проблемасы білімді пайдалануды ғана емес, оны толықтырып өзгертіп тұру техникасын да қолдануды қажет етеді. Бұл мәселе оқытып, үйрету мәселесімен бірте қайнасып жатады. Үйрену функциясы тірі табиғаттағы барлық тірі жандарға тән. Сондықтан осыларға тән үйренуді бақылау және үлгілеу, математикалық немесе таңбалы логика көмегімен үйрену үлгілерін зерттеу, іс жүзіндегі бейнелерді өңдегендегі үдерістерді үлгілеу сияқты мәселелер үйрету функциясын зерттегенде қарастырылады. Үйрену мәселесін үш түрлі бағытта қарастыруға болады. Біріншісі – адамның ойлау мен үйрену функцияларын зерттеу және оны компьютерде үлгілеу, келесісі – ақпараттық логикалық үйрену функцияларын зерттеу, үшіншісі – бейнелерді өңдеуде үйрену үдерісін бақылау.

Қорытындылай келсек, оқу құралында жасанды зерде аумағына қатысты есептерді шешудің үлгілері мен әдістеріне, сарапшы жүйе мәселелеріне жататын зерттеулерге талдау жасалған. Сонымен бірге кітапта программалаудың функционалдық және логикалық тілдерге қатысты негізгі мәселелері қарастырылған. Аталған программалау стилінің негізгі өкілдері Лисп, Хаскелл және Пролог тілдерінің негізгі ұғымдары мен мүмкіндіктері келтірілген.

Қазіргі заманда адамзат баласы болашағына үлкен мақсаттар орындалуын күтетін жасанды зерде бағытының мәселелеріне қасиетті қазақ тілінің биігінен ұмтылу арқылы ұрпағымыз адамзат өркениетінен қалыспай, басқа елдердің ұлдары мен қыздарындай, ғылым көкжиегінің биігінде қалықтайды деген үміттемін.

1-ҚОСЫМША

Нейропакеттер кестесі

Атауы, мекені	Өндіруші мекені	Платформасы	Үйрету алгоритмдері	Интерфейсі	Түсіндірулер
1	2	3	4	5	6
Matlab Neural Network Toolbox 3.0	MathWorks, США http://www.mathworks.com	Win 95, 98, NT 4.0	Қолданудағы парадигмалар: персептрон, кері шақыру, радиалды базис, Эльман желісі, Хопфилд желісі, ықтималды регрессия	Ms Windows арналған GUI	ANSI жарайтын коданы генерациялайды
SNNS http://www.informatik.unistutgart.de/ipvr/bv/projekte/snns/announce.html	Штудтгард университеті, Параллель және үлестіру жүйелері Институты (IPVR)	Unix	Кері шақыру, радиалды базис, Эльман желісі, ART1, ART2, Кохоиен карталары, Джордан желісі, ассоциативті жады	X-Windows арналған GUI	Ең жақсы симулятордың бірі. MS Windows ортасында X-Windows эмуляторымен бірге жұмыс істейді. C++ тілінде бастапқы кодалары бар
Trajan Trajan Software Ltd.	Ұлыбритания http://www.trajan-software.demon.co.uk/commerce.htm	Win 3.x, 9x, NT	Көпдеңгейлі персептрон, кері шақыру. Кохонен карталары, радиалды базис, ықтималды регрессия	Ms Windows арналған GUI	Демо нұсқасы өндіруші компания сайтында бар
Delta	Artificial Intelligence Group, Франция http://wwwinf.enst.fr/~milc/dnns/dnns.us.html	HP-UX Sun OS 4.1	Көпдеңгейлі персептрон, кері шақыру. Кохонен карталары, Джордан желісі, Ельцин желісі	X-Windows арналған GUI	Демо нұсқасы бар

кесте жалғасы

1	2	3	4	5	6
X-Sim	IIC,Испания (Мадрид) http://www.iic.uam.es/xsim/Welcome.html	Unix Linux	Көпқатпарлы персептрон, кері шақыру, Кохонен карталары	Unix, X-Windows арналған GUI	Демо нұсқасы бар
Brain Wave	Университет Куинсланд, АҚШ http://www2.psy.uq.edu.au/~brainwav/		Көпқатпарлы персептрон, кері шақыру, Кохонен карталары, Хэббиан желісі	Javaны колдайтын Internet-броузер	Java-апплета түрінде іске асырылған, кез келген операциялық жүйеде істейді
VieNet2 www.ai.univie.ac.at/oefai/nn/tool.html	Жасанды зерде мәселесін зерттейтін Австрия институты	DOS Win Unix Linux	Көпқатпарлы персептрон, кері шақыру, Кохонен карталары, Джордан желісі, Эльман желісі, ассоциативті жады	Пайдаланушы қалыптастырады	Бастапқы кода түрінде тарайды, жеке программаны құруға ыңғайлы
NeuroWindows	НейроЖоба, Россия http://www.neuroproject.ru/	Win	Кері шақыру, Кохонен карталары, ассоциативті жады	Пайдаланушы қалыптастырады	Visual және Delphi кітапханасы бар
Aspirin/ MIGRAINES	Mitre Corp.	Unix	Кері шақыру	X-Windows арналған GUI	Нейрон желісінің салмақтарын, вектор түйіндерін дискіде сақтайды
Atree alnl@cs.ualberta.ca	Билл Армстронг, Альберта, АҚШ	Dos, Unix	Адаптивті логикалық ағаштар	Unix, Dos терезелер	Демо нұсқасы бар
Cnaps	Adaptive Solutions Inc.	SunOS	кері шақыру, Кохонен карталары, LVQ2 және жиілік-сезімдік оқыту	X-Windows арналған GUI	Кері шақырумен үйретуде жылдам-дық 1 биллион CUPS

кесте жалғасы

1	2	3	4	5	6
ICSIM	Халықаралық компьютер ғылымдарының институты, Беркли, Калифорния, АҚШ	Unix	Алдын ала анықталған желілер	Shell, X-Windows арналған GUI	Демо нұсқасы бар
Neural Shell ftp://ftp.quanta.eng.ohio-state.edu/	SPANN, зертханасы Огайо, АҚШ	Unix	Хопфилд желісі, Хемминг желісі, кері шақыру, Кохонен карталары, адаптивті баяу кері шақыру, жиіліксезімдік оқыту	SUNTOOLS және X-Windows арналған GUI	Демо нұсқасы бар
Neuron	Университет Дьюка, АҚШ	Unix	Үшөлшемді пирамидалық ұяшық, диффузия	X-Windows арналған GUI	Демо нұсқасы бар
Sankom	Дортмунд университеті, Германия	Unix	Кохонен карталары	Shell, командалық жол	Демо нұсқасы бар
SOMPAK	SOM, компьютерлік, ақпараттық ғылым зертханасы, Хельсинки технология университеті	Dos, Unix	Өзімен-өзі ұйымдасатын карталар	Пайдаланушы қалыптастырады	Демо нұсқасы бар
Xerion	Университет Торонто,АҚШ ftp://ftp.cs.toronto.edu/pub/xerion	Unix	Кері шақыру, рекурентті кері шақыру, Больцман машинасы орта өріс теориясы, Кохонен карталары	X-Windows арналған GUI	Демо нұсқасы бар
NETS service@cossack.cosmic.uga.edu	COSMIC, Университет Джорджии, АҚШ	Dos, Unix	Кері шақыру	Командалық жол	Демо нұсқасы бар

ГЛОССАРИЙ

1. **Антецедент** – екі орындық логикалық операцияның алғышарты немесе импликацияның сол жағы.

2. **Аппликация** – Аппликация конструкциясын енгізу өзінің екі аргументін (оператор және операнд) бірдей есептейтін функцияларды есепке алуды жеңілдетеді. Ол *f(a)* функциясын *f* және *a* аргументтеріне аппликация (қолданба) операциясын қолдану нәтижесі деп қарайды. Ол былайша өрнектеледі: $f(a) \leftrightarrow apply(f,a)$ де, осы операцияны *аппликация* деп атайды.

3. **Арндық** – есептеу предикаттарында қолданылатын функционалдық таңба. Функционалдық тәуелділікте көрсетілген параметрлердің санын көрсетеді.

4. **Атом** – есептеу предикаттарында қолданылатын қарапайым формула. Ол «шын» немесе «өтірік» мәнін қабылдайды.

5. **Ассоциативті жады** (Ассоциативная память) – машина жадысының (зердесінің) ерекше түрі. Ол ақпаратты өте тез іздеп табу үшін қажет. Сонымен бірге компьютердің мазмұны бойынша іздеу зердесі, контент мекені бойынша іздеу зердесі сияқты түрлері де бар.

6. **Атомдық формула** – есептеу предикаттары тілінде қолданылатын ең қарапайым бірлік. *N* арндық предикат болып табылады, одан соң өзара үтірлермен бөлінген жақшаларға орналасқан *n* термдер орналасады.

7. **Бар болу кванторы** – бұл ұғым ∃ таңбасымен белгіленеді. (теріс түскен Е әрпі). Келесі $\exists Y$ өрнегі мынаны білдіреді: ең болмағанда *Y* айнымалысының бір мәні үшін одан кейінгі келетін сөйлем шын мәнінде болады.

8. **Бинарлық резолюция** – бір литер, екіншісі оның терістеуі болатын екі өрнекке қолданылады.

9. **Бектрекинг (backtrack)** деп аталатын қайтаруы бар іздеу тәсілін атауға болады. Бұл тәсіл есептің мүмкін болатын шешімдері көп болатын жағдайда қолданылады. Бектрекинг тәсілінің мәні мынада: есептің іздеуінде тармақталатын жолдарда оның біреуімен кетіп, онда мақсатты күй болмаса, қалған жолдарға қайтып келу үшін осы тармақты еске сақтап қою. Жалпы жағдайда мұндай қайту тармақтары көп болуы да мүмкін. Оларды *бектрекинг нүктелері* немесе *тармақтар* деп атайды. Пролог тілінің кейбір нұсқаларында және Плэнер тілінде осы бектрекинг тәсілін іске асыратын арнайы механизмдер бар.

10. **Білім инженері** – білімдерді бейнелейтін және ЖЗ тілдерін меңгерген маман. Оның басты мақсаты – құрылатын жоба үшін

программалық және аппараттық құралдарды таңдау, сала мәселесіне қатысты оның маманының қажетті ақпаратын түсінікті түрге қалыптастыру және осы білімдер жиынын білім қорына тиімді және анық түрде орналастыру.

11. **Білімдер қоры** – кез келген сала мәселесіне қатысты білімдерді бейнелейтін үлгілер жиынтығы. Онымен қатынайтын тілдер тәуелсіздігін қамтамасыз ету арқылы ұйымдастырылған.

12. **Верификация** (лат. verus – «шын» және facere – «жасау») – алгоритмдерді, программаларды, процедураларды олардың тәжірибелік түрде алынған эталондармен, нұсқаларымен салыстыруды, тексеруді, тұжырымдауды іске асыратын тәсіл.

13. **Демон** (ағылш. daemon) – Бұл ұғым фреймнің құрылымдық элементтерінің бірі. Ол белгілі бір шарт орындалғанда іске қосылатын процедура. Демондар арнайы бір слотқа қатынағанда іске қосылуы мүмкін. Демондардың бірнеше түрі бар. Пролог тілінде IF-NEEDED демоны слотқа қатынау кезінде оның мәні анықталамаған жағдайда іске қосылса, ал IF-ADDED демоны слотқа белгілі бір мән берілгенде IF-REMOVED демоны слот мәнін өшіргенде іске қосылады. Демондар осындай қосақталған процедуралар механизмдері жағынан реляциялық дерекқорлардағы триггерлер процедураларына ұқсас болып келеді.

14. **Деректердің енгізілген түрі (встроенные типы)** – ЖЗ тілдерінде кездесетін деректер типі, мысалы, Лисп тілінде кездеседі. Бұл тілде типтер айнымалыға қатысты емес, деректер нысандарына қатысты анықталады. Анықталған кез келген нысанды программа орындалып жатқанда кез келген айнымалымен байланыстыруға болады. Осындай икемділікті қамтамасыз ету үшін, Лисп тілінде программа орындалып жатқанда типтер сәйкестігін тексеру іске асырылған.

15. **Деректердің ішкі типтері** – Лисп тіліне тән деректер типтері. Лисп тілінде типтер айнымалыларға емес, деректер нысандарына жатады. Программа орындалғанда кез келген нысанды кез келген айнымалымен байланыстыруға болады. Осы икемділікті ұстап тұру үшін, Лисп тілінде программа орындалғанда типтер сәйкестігін тексеру болады.

16. **Генетикалық алгоритм** – оңтайлы биттік қатарды іздеу алгоритмі. Ол ең алдымен осындай қатарлар популяциясын кездейсоқ түрде таңдайды, содан соң оны жасанды өзгеріс үдерісіне апарады. Бұл үдеріс табиғи түрдегі сияқты таңдау мен қосақтау үдерістерінен тұрады.

17. **Жазба қысқартылуы (упрощение записи).** Функционалдық программалауда *apply* операциясы анық түрде көрсетілмейді де, оның

орнына мына жазба түрлері қолданылады: *((x)(y))* немесе *(x y)* және егер жақшаларды алып тастасақ жазба түрі *x y* болады. Жақшалардың санын азайту мақсатында мынандай келісім болған: $a_1\ a_2\ \ldots\ a_n \leftrightarrow (\ldots((a_1\ a_2)\ a_3)\ \ldots\ a_n$, мұндағы жақшалар келісім бойынша солдан оңға қарай орналасады.

18. **Жалпылық кванторы** мына ∀ таңбамен белгіленеді (теріс түскен А әрпі). Квантор артынан *X* айнымалысы тұрады. ∀*X* белгісі мынаны білдіреді: бұдан кейінгі сөйлем *X* айнымалысының барлығы үшін шын мәнінде болады.

19. **Жасанды зерде** – компьютер ғылымдарында саналы түрдегі тәртіпті істеуді автоматтандыруға арналған аумақ.

20. **Жасанды нейрон** – жасанды *нейрон,* негізінен, биологиялық нейрон қасиеттерін қайталайды. Жасанды нейронға енетін сигналдар жиынының әрбіреуі басқа нейронның шығу сигналы болып табылады. Әр ену сигналы синаптикалық күші бар салмақ шамасына көбейтіліп, осы көбейтінділердің барлығы бір-бірімен қосылады. Осы қосынды нейрон белсенділігінің деңгейін анықтайды.

21. **Жасанды нейрон желісі** қарапайым желі қатпарларын құрайтын нейрондар тобынан тұрады, оны Бір қатпарлы жасанды нейрон желісі деп атайды. Көп қатпарлы желілер қатпарлар каскадынан құрылады. Бір қатпардың шығуы келесі қатпар үшін ену болып табылады.

22. **Жасанды нейронның белсенді функциясы** – электрондық жүйелерге ұқсас белсенді функцияны жасанды нейронның сызықтық емес түрдегі күшейткіш сипаты есебінде қарастыруға болады.

23. **ЖӘНЕ/НЕМЕСЕ граф** – *ЖӘНЕ* мен *НЕМЕСЕ* операторларын граф көмегімен бейнелеу тәсілі. Ол жасанды зерде мәселелеріндегі іздеу жолдарын сипаттауға қажетті маңызды құралдардың бірі болып табылады.

24. **ЖӘНЕ/НЕМЕСЕ графы** – логикалық ЖӘНЕ операторы мен НЕМЕСЕ операторларын граф арқылы бейнелейтін тәсіл. Ол – жасанды зерде мәселелеріндегі іздеу аумағын сипаттауға арналған маңызды құрал.

25. **Жиындар** – бұл ЖЗ тілдерінің қайталанбайтын элементтер жиыны, мысалы, Пролог тілі үшін. Оны барлық балалық күйлерді біріктіру үшін немесе closed тізімін ұстап тұру үшін пайдаланады. Элементтер жиыны, мысалы, *(a, b)* егер ондағы элементтер ретінің әсері болмаса, *[a, b]* тізімі түрінде пайдаланылады.

26. **Иерархиялық фрейм.** Әдетте, фреймдердің құрылымы иерархиялық түрде сипатталып беріледі. Мұндай иерархиялық

құрылымдардың ерекшелігі мынада: ең жоғарғы деңгейдегі фреймге қатысты атрибуттардағы ақпаратты осы фреймге кіретін төменгі деңгейдегі басқа фреймдердің бәрі пайдалана алады.

27. **Импликация** – екі орындық логикалық операция. $P \rightarrow Q$. Мұндағы P – алғышарт немесе антецедент, Q – қорытынды немесе логикалық соңы.

28. **Индуктивті шек** – үйрету жүйелеріне арналған таңдау критерийі. Үйрету жүйелері ақпаратты эвристикалық түрде өңдейді, яғни болашақта пайда әкеле алады-ау деген сенімділікті ақпаратты өңдейді.

29. **Интерпретация** – логикалық сөйлемдерге шындық мән беру. Белгілі бір бар дүние үшін шын болатын тұжырымдар үшін орындалады.

30. **Каррирлеу (каррирование).** Бұл тип атауы есептеу үдерісінің сипаттауын комбинаторлық логикаға негіздеушілердің бірі Хаскелл Карридің атымен аталған. Бұл жағдайда функция аргументі не онын мәні есебінде басқа функция бола алады. Көптеген аргументтері бар функцияны бір аргументті жоғарғы тәртіпті функция есебінде көрсете аламыз.

31. **Кезек (queue)** – дегеніміз бұл деректер құрылымы. Оны әдетте [П. О] (First-In-First-Out) – "бірінші кірді, бірінші шықты" құрылымы деп атайды. Бұл құрылымды да тізім есебінде карауға болады. Ондағы элементтер бір жағынан алынып, екінші жағынан қосылады. Кезек жайылып іздеу алгоритмінде қолданылады.

32. **Кері тарату алгоритмі** (backpropagation learning **rule)** – көп қатпарлы персептрондарды үйрету алгоритмі. Кеңінен таныс, сенімді алгоритм. Бірақ нейрон желілерінің басқа алгоритмдеріне қарағанда баяу.

33. **Клоз** (ағылшын. clause) дегеніміз көрсетім түрлерімен анықталған белгілі бір функцияның есептеу нәтижесінің бір нұсқасының жазбасы болып табылады. Клоздың Бэкус-Наур нотациясы бойынша жазбасы келесі түрде болады: *<clause > : : = <faunction_name > <patterns > = <expression>* мұндағы *faunction_name* – функция атауы; *expression* – тіл синтаксисі тұрғысынан дұрыс құрылған белгілі бір өрнек; *patterns* – салыстырылуға арналған көрсетім түрлерінің тізбектері.

34. **Кохонен желілері** – адам миының ақпаратты топологиялық түрде қабылдау қасиетін қайталайтын нейрондық желілер. Оларды сонымен бірге өзін-өзі ұйымдастыра алатын белгілеу карталары деп те атайды (SOFM).

35. **Көрсетім үлгілері (ағылш. pattern)** деп деректерді салыстыруға пайдаланылатын оларды құрастыру операциясының өрнектерін айтамыз. *Көрсетім үлгілеріне* қойылатын талап біреу: деректерді көрсетім түрлерімен салыстырғанда, көрсетімге енетін айнымалыларды *таңбалау (означивание)* тек бір-ақ рет болады.

36. **Куиллиан үлгісі.** Бұл үлгідегі тұжырымдамалық нысандар ассоциативтік торлар түрінде беріледі. Тор өз төбесіндегі ұғымдар, оларды байланыстыратын доғалар-концептер арасындағы қатынастарды білдіреді. Куиллиан құрған семантикалық тор үлгісінің маңыздылығы мынада: әдетте, қолданылатын құрылымды талдайтын семантикалық өңдеу әдістерінен тыс, ұзақ уақыттық жады құрылымы ұсынылады. Бұл құрылымда тілдік белсенділік көлеміне көп мән беріледі. Фактілер мен ұғымдар арасындағы қатынастарды сипаттайтын тәсіл ұсынылады. Бұл тәсіл семантикалық торлар деп аталады. Мұндай семантикалық өңдеу тәсілі *Мир понятий – ұғымдар әлемі* деп аталатын аумақта іске асады. Бұл *ұғымдар әлемі* – өзінің бойында белгілі бір алфавитті, ұғымдарды, ережелерді аксиомаларды жинақтаған нақты формальды теория.

37. **Кросс-тексеру** – нейрон желілеріне қатысты итерациялық оқыту кезінде бақылау жиыны деп аталатын қосымша деректерді пайдалану. Деректердің үйрету жиыны желі салмақтарын жөндеуге арналса, ал бақылау жиынындағы деректер нейрон желісінің ақпаратты жинақтауды қалай үйренгенін тексеруге қажетті болады.

38. **Қорғау (охрана).** Қорғау шарты мына *when* және *otherwise* негізгі сөздермен енгізілетін логикалық өрнек арқылы анықталады. Сандық тізімдерге белгілі бір операцияларды орындағанда, сан оң не теріс болса, әртүрлі амалдар орындау қажеттігі туады. Осы жағдайда клоздың оң жағында шартты өрнекті қолдануға болады. Осы механизмді *қорғау* деп атаймыз.

39. **Лисп тілі.** Лисп тілін ең алғаш рет Джон Маккарти (John McCarty) 1950 жылдың аяғында ұсынған. Ең алғашында бұл тіл рекурсивтік функциялар теориясына негізделген есептеу үлгісі ретінде қарастырылды. Бұл тіл сандық есептеулер емес, таңбалармен жұмыс істеуі керек. Лисп тілі – жасанды зерде тәсілімен құрылатын программалар үшін ең тиімді де қажет құрал. Лисп тілі трансляторының бірнеше түрі бар. Лисп тілінде машина жады былайша пайдаланылады: егер жаңа ұғымды жадыға орналастыру жағдайы қажет болғанда, машина зердесінен осы ұғымға қанша орын керек болса, сонша беріледі.

40. **Лямбда-өрнектер (лямбда-выражение)** – әдеттегі математикалық мәтіндерде функцияны анықтау былайша өрнектеледі: $f(x) = a + bx$,

А.Чёрч анықтамасы бойынша функцияны келесі түрде жазамыз: $f = \lambda x.a + bx$ мұндағы теңдеудің сол жағында f – идентификатор, оң жағындағы өрнек *лямбда-өрнек* деп аталады. Лямбда-өрнек құрамында «λ» таңбасы бар, одан кейінгі орналасқан x айнымалысын байланған айнымалы деп, ал « . » таңбасынан кейінгі өрнекті лямбда-өрнектін денесі деп атайды.

41. **Метабілімдер** – білімдер жайлы білімдер. Зерделік жүйе сала мәселесін қана біліп қоймай, ол туралы өзінің білетінін де білуі керек. Бұл «өз білімі туралы мағлұматтың болуы» білімдердің жоғары деңгейін береді және зерделік жүйелерді жобалауға және оны толық сипаттауға қажет болады.

42. **Минимакс** – сандар жиынтығын сызықтық масштабтау коэффициенттерін анықтау алгоритмі. Ең аз және ең үлкен мәндер таңдап алынады, содан соң масштабтау коэффиценттерін таңдау жүргізіледі. Олар түрленген деректер жиынтығының алдын ала ең аз, не ең көп мәні болатындай таңдап алынады.

43. **Мультийерархиялық үлгі** – бұл үлгідегі интерфейс – *тілдер* аумағы, ал ішкі білімдер белгісі ұғымдар аумағы болып табылады. Түсінік аумағындағы ұғымдар ең алдымен *грамматика* әлемінде талданады. Содан соң түсінік бөліміндегі ұғымдар бөлініп алынып (топ немесе жеке), *Сөздік* әлемінде талданады.

44. **Нейрондық торлар** – байланыс торлары. Олардағы үйрету таңбалы тіл негізінде өтпейді. Тірі организмдер миындағы көптеген нерв клеткаларына ұқсас, нейрондық торлар – ол өзара байланысқан жасанды нейрондар жүйелері. Компьютер программасының білімінде осы нейрондардың өзара әсерлері мен олардың жалпы ұйымдасқан түрлері бар.

45. **Неокогнитрон** – мини компьютер негізінде іске асырылған тәжірибелік жүйе. Ол арнайы енгізу құрылғысы көмегімен жазылған таңбаларды тани біледі. Таңбаларды таныпүйрену үдерісі бірнеше секундқа созылады. Бұл жүйенің сыртқы белгі бойынша жүйелей білу қасиеті ғана емес, сонымен бірге оның үйрене білу қабілеті де бар. Яғни Неокогнитронға бір таңбаны бірнеше қайтара көрсетіп, жүйенің бұл таңбаны тани білу реакциясы тұрақты болғанға дейін және бұл үдерісті келесі тануға қажет таңбаға өткенде қайталаса, онда жүйе бейнелерді тани білу қабілетіне ие болады. Бұдан ары қарай үйрену үдерісіне кері байланыс тізбегін қоссақ, онда жүйе ассоциативтік түрде бейнелеу қабілетін үйренеді.

46. **Оқыту жылдамдығы** – нейрон желілеріндегі басқарушы параметр. Кейбір оқыту алгоритмдерінде салмақтарды итерациялық жөндеуде қадам шамасын бақылайды.

47. **Оқытушымен үйрену** – нейрон желілерінде оны кейде басқарылатын оқыту деп те атайды. Нейрон желісінің енуіне белгілі шығу айнымалылары бар деректерді беретін алгоритм түрі. Желідегі салмақтар нақты алынған шығу мәндерімен салыстыру нәтижесінде жөнделіп отырады.

48. **Оқытушысыз үйрену** – нейрон желісіндегі салмақ коэффициенттерін жөндеу оның алдындағы нейрондардағы және алдыңғы коэффициенттер мағлұматтары негізінде болып отырады.

49. **Персептрон** – мұғалімдік үйрету жүйесі. 1958 жылы Розенблатт бейнелерді танып үйренудің, үш қатпарлы құрылымдық жүйкелік тізбелер үлгісіне негізделген жүйені құрды. Бұл жүйенің жұмыс істеу ерекшелігі мынада: бірінші – қатпардағы бірнеше элементтерден шығатын сызық байланыстар, екінші – қатпардағы бір элементте тоғысуы мүмкін.

50. **Предикат** – есептер дүниесіндегі бірнеше нысан арасындағы қатынастарды көрсетеді. Осылайша байланысқан нысандар саны предикаттың «арность» деп аталатын өлшемін береді.

51. **Предикаттарды есептеу** – жасанды зерде есептерін сипаттауға арналған тіл. Зерделі және сарапшы жүйелердің тұжырымдаудағы сапалы аспектілерін бейнелейтін тәсіл.

52. **Продукциялық жүйе** – С деп хабарлар құрылымына әсер ететін ұйымдастырылған немесе жекеше түрдегі программа жиындарының белгілі бір әдіспен ұйымдастырылған есептеу іс барысын айтамыз. Бұл программалар: Орындалу шарты С құрылымына деген кейбір талап-тілектерден тұрса, ал іс-әрекет осы талаптар орындалған жағдайда, істелуге тиісті жұмысты білдіреді.

53. **Пролог тілі.** Бұл тіл – логикалық программалаудың ең көп мәлім болған мысалы. Пролог 1-ші дәрежедегі предикаттар теориясына негізделген. Бұл тілдің атының өзі былайша таратылады: Programming in Logic (Логикадағы Программалау). Программа орындалғанда интерпретатор үнемі логикалық спецификациялар негізінде шешімді шығарып отырады. Пролог тілінің даму тарихы теоремаларды дәлелдеу, оның ішінде резолюциялық терістеу алгоритмін жасау кезеңінен бастау алады. Бұл алгоритмде *резолюция* (resolution) деген атқа ие болып, кейіннен Пролог тілінің есептеулеріндегі негізгі әдіс болып кеткен дәлелдеу амалы келтірілген.

54. **Резольвента** – резолюция көмегімен алынған нәтиже қарама-қайшылыққа әкелмейтін шешім түрі.

55. **Резолюция** – жасанды зерде саласына қатысты предикаттарды немесе тұжырымдарды есептеу теорема саласындағы қолданылатын тәсілдердің бірі. Терістеуді құруға пайдаланылатын ереже.

56. **Рекурсия** – белгісіз өлшемді, дұрыс құрылымды деректер жиынын басқаруға арналған құрал. Мысалы, тізімдер, ағаштар, графтар. Рекурсия құралы кеңістіктегі күйі бойынша іздеуде өте ыңғайлы.

57. **Салмақ коэффициенті** – нейрондар арасындағы байланыс күшін анықтайтын коэффициент.

58. **Сарапшы жүйе** – белгілі бір сала мәселелерін шешуде сарапшы-маман адамды ауыстыра алатын жасанды жүйе. Бұл жүйе жөніндегі зерттеулердің мақсаты – құрылымы қиын есептерді шығарғанда сапасы мен тиімділігі жағынан адам тапқан шешімнен кем түспейтін нәтижеге жететін компьютерлік программалар жасау.

60. **Семантикалық торлар** білімдерді граф түрінде бейнелейді. Оның түйіндерінде фактілер мен ұғымдар, ал доғаларында осы ұғымдар арасындағы байланыстар мен ассоциациялар көрсетіледі. Әдетте, түйіндер де, байланыстар де таңбаланады. Білімдерді көрсетіп, пайдалану тәсілдерінің біріне *семантикалық тор* түріндегі үлгіні қарауға болады. Мұндай үлгі түрі психологияда «ұзақ уақыттық жады (ес)» деп аталатын құрылымды сипаттайтын тәсіл түрінде белгілі. Бұл тәсіл кейіннен білімді пайдалану үлгілерінің негізгі түрлеріне енді. «Семантика» деп сөздің, сөйлемнің, шығарманың, іс-әрекеттің, жағдайлардың мән-мағынасын анықтайтын ұғымды айтамыз. Басқаша айтқанда, «Семантика» деп таңбалар мен осы таңбалар жиынтығынан тұратын нысандар арасындағы белгілі бір қарым-қатынасты айтуға болады. Ал «Прагматика» дегеніміз – сөз бен таңба құрушылары арасында болатын айшықты қарым-қатынас түрі. Семантикалық торлардың білімді пайдалану үлгілерінің басқасынан негізгі ерекшелігі – ол жүйенің бүтіндігін қажет етеді. Яғни білім қорындағы білімдер мен оны шығару механизмі бір-бірінен бөлініп қаралмайды.

61. **Синапс** – нейрондар арасындағы байланысты білдіретін шама. Осы байланыс күшін көрсететін параметрді синапс салмағы деп атайды. Салмақтарының мәні оң болған байланыс түрін қоздырушы, ал теріс болған байланыс түрін тежеуші деп атайды.

62. **Стек (stack)** – қатынау мүмкіндігі тек бір жағынан ғана іске асатын сызықтық құрылым. Демек, құрылымның барлық бөліктері бір ғана жағынан қосылады немесе өшіріледі. Стекті «П..О» құрылымы

(Last-In-First- Out) – «соңғы енді, бірінші шықты» деп атайды. Осындай құрылым мысалы есебінде тереңнен іздеу алгоритмін атап өтуге болады. Стек жұмыс істеуі үшін келесі операциялар анықталуы қажет.

63. **Сценарийлер** – белгілі бір мәтінде уақиғаны тізбектеп баяндау. Табиғи тілді түсіну жүйелерінде жүйе оны түсінетіндей жағдай терминдерімен білім қорын ұйымдастыру үшін қолданылады.

64. **Терм** – предикаттарды есептеу теориясының таңбасы. Ол арқылы есепті қою аумағындағы нысандар мен олардың қасиеттері белгіленеді.

65. **Тізім** – бұл реттелген элементтер жиынын деректер құрылымы (немесе тізімдердің өздері де реттеледі). Рекурсия – осындай тізімдік құрылымдарды өңдейтін табиғи тәсіл. Тізім бейнелеу құрылымдарын жасауға арналған өте икемді құрал. Тізімдер элементтері квадрат жақшаға [] алынып, бір-бірінен үтір арқылы бөлінеді.

66. **Торлық үлгілер.** Бұл үлгілер тобының негізінде төбелері ұғымдарды білдіретін, ал оларды қосып тұрған сызықтар – доғалар, олардың арасындағы байланыстарды білдіретін тор ұғымы жатыр. Төбелер мен доғаларды бейнелеп түсіндіруге белгілі бір шектер қоя отырып, тордың әртүрлі түрін алуға болады. Егер торлар төбелеріндегі ұғымдар құрылым жағынан қарапайым болса, онда оларды – *қарапайым*, ал ұғымдардың ішкі құрылым күрделі түрде кездессе, онда оларды *иерархиялық* түрдегі торлар деп атайды. Төбелер арасындағы байланыстар бір түрде болуы мүмкін, мұндай жағдайда торлар *біркелкі* деп аталады. Егер төбелер арасындағы байланыс түрлері әртүрлі мағынада болып келсе, торлар *әркелкі* деп аталады. Торлық құрылыммен берілген үлгілердің төрт түрін атап өтуге болады. Оған *функционалдық, сценарийлер, семантикалық торлар*, фреймдер жатады.

67. **Тізімдік құрылымдар.** Лисп тілі үшін. Тізімдік элементтердің бәрінің типтері біреу болады. Мысалы, *А*-типі. Ол былайша белгіленеді: *ListStructure(A)*. Бұндай тізімдік құрылымдар үшін *«ену деңгейі»* деп аталатын ұғым бар. Ол ең көп «*(*» ашылу жақшасының санына тең болады.

68. **Тұжырымдамалық граф** – соңғы, байланған, екі жолды граф. Граф түйіндерінде ұғымдар немесе тұжырымдық қатынастар орналасады. Ұғымдар арасындағы байланыстар доғалармен беріледі.

69. **Тұжырымдық үлгі** – сала мәселесіне қатысты білімдер тұжырымдамасы. Оны білім инженері құрады. Бұл үлгі қалыптасқан білім қорының құрылымын анықтайды.

70. **Унификация** – предикаттарды есептеу теориясындағы екі өрнекті бір-біріне сәйкестіруге қажетті алмастыруларды анықтайтын алгоритм.

71. **Фреймдер** – Сала мәселесіне қатысты анық берілмеген ақпараттық байланыстарды қатаң түрде ұйымдастырылған құрылымдармен бейнелеу үшін қолданылатын сызбанұсқа.

72. **Фреймаралық желілер.** Классификациялық иерархиялық құрылымы бар тұжырымдамалық нысандарды іздегенде іздеу үдерісі сәтсіз аяқталса, онда осыған ұқсас тағы басқа фреймді іздеу қажеттігі туады. Бір-бірінен бөлектегіш нұсқағыштары арқылы байланысқан *фреймдер желісін* құра алады. Егер, мысалы, «стол», «орындық» деген ұғымдар фреймдерін бөлектегіш нұсқағыштармен байланыстырсақ, *фреймдер желісін* аламыз. «Стол» деген фреймнің бөлектегіш нұсқағышына «арқалығы жоқ», «тамақ ішуге, жазуға, оқуға арналған» деген ұғымдар жатса, «Орындық» фреймінің бөлектегіш нұсқағышына «столдан аласа», «отыруға арналған» деген ұғымдар жатады.

73. **Функционалдық тип** – f функциясын қарастырғанда, оның айнымалысының анықталу аумағы А типіне, өз мәндерінің анықталу аумағы B типіне жатады делік. Сонда функция типі мына өрнекпен $A \rightarrow B$ анықталса, онда оны *функционалдық тип* деп атаймыз.

74. **Функционалдық түрлер** – функционалдық программалаудың математикалық негіздерінде белгілі бір атаулар қалыптасқан. Бұл атаулардың функциялық түрлеріне: функционалдық тип, каррирлеу, аппликация, өрнекті жалпылау, жазба қысқартуы, лямбда-өрнектер жатады.

75. **Хаскелл тілі.** Haskell тілі – бұл жалпы мақсаттарды орындайтын қазіргі заманғы тіл. Ол функционалдық программалауды қолдайтындардың ұжымдық білімдерін жинақтап, қуатты, жалпыға бірдей, көркем тілді құру мақсатын іске асырған тіл. Бұл тілдің өзіне қатысты бірнеше қасиеттері бар. *Тазалығы.* Онда ешқандай қосымша әсерлер жоқ. *Қалдырылған (жалқау) есептеулер.* Ол есептеуге қажеттілік тумаса, ол есептелмейді. *Қатаң типтеу.* Бұл тілде байқаусызда *Double* типін *Int* типіне келтіру мүмкін емес және бос көрсеткіш бойынша қажет әдісті де шақыра алмайсыз. Бұл қателер санын азайтуға әсерін тигізеді.

76. **Эвристика** – бұл кеңістіктегі күйі бойынша таңдап іздеудің стратегиясы. Ол іздеуді шешім бар-ау деген жолға бағыттайды. Ол үшін эвристика – сезім мен түйсіктен туындаған алгоритмдерді пайдаланады.

77. **Ықтималды нейрон желілері (PNN)** – жүйелеу есептеріне арналған нейрон желілерінің түрі. Ықтималдылық тығыздығы белгілі бір топтарға жатады.

Әдебиеттер тізімі

1. Доорс Дж. И др. Пролог – язык программирования будущего /Пер. с англ. – М.: Финансы и статистика, 1990. – 144 б.
2. Братко И. Программирование на языке Пролог для искусственного интеллекта. М: Мир, 1990 ж.
3. Язык Пролог в пятом поколении ЭВМ. Сборник статей под редакцией Ильинского. М: Мир, – 1988. – 154 б.
4. Клоксин, Меллиш «Программирование на языке Пролог». М: Мир, 1987 ж.
5. Дж. Стобо «Язык программирования Пролог». М: Радио и связь. 1993 ж.
6. Хоггер «Введение в логическое программирование». М: Мир, 1988 ж.
7. Л. Стерлинг, Э. Шапиро «Искусство программирования на языке Пролог». М: Мир, 1990 ж.
8. http://revolution.allbest.ru/programming/00138284.html.
9. Люгер, Джордж Ф. Искусственный интеллект: стратегии и методы решения сложных проблем. М:, 2003 ж. – 864 б.
10. Джексон Питер, Введение в экспертные системы. / Пер. с англ.: Уч. пособие, 2001 ж. – 624 б.
11. Хъювенен Э., Сеппянен Й. Мир Лиспа В 2-х т. Т.1: Введение в язык Лисп и функциональное прогаммирование. Пер. С финск. – М.: Мир, 1990. – 447 б.
12. Хъювлеен Э., Сеппянен Й. Мир Лиспа В 2-х т. Т.2: Методы и системы прогаммирования. Пер. с финск. – М.: Мир, 1990. – 319 б.
13. Симанков, В.С. Основы функционального программирования – Краснодар: КубГТУ, 2002. – 160 б.
14. Сергиевский Г.М. Функциональное и Логическое программирование. – М.: Изд. Центр «Академия». – 2010. – 320 б.
15. http://revolution.allbest.ru/programming/001181227.html
16. http://ru.wikibooks.org/wiki/ _Haskell
17. http://haskell.org/, http://www.haskell.ru/
18. http://haskell.org/hugs)
19. http://wolfram.com)
20. Mathematica (http://wolfram.com)
21. «http://ru.wikibooks.org/wiki/_Haskell»
22. Журнал «Потенциал», № 1, 2006.
23. http://roman-dushkin.narod.ru/fp.html.
24. Нильсон Н. Искусственный интеллект. Методы поиска решений. / Пер. с анг. – М.: Мир, 1973. – б. 270
25. Представление и использование знаний / Пер. с японск. /Под ред. Х.Уэно, М.Исидзука, – М.: – Мир, 1989. – 220 б.

26. Приобретение знаний. / Пер. с японск. /Под ред. С.Осуги, Ю.Саэки М.Исидзука, – М.: – Мир, 1990. – 304 б.
27. Ахметова М. Сарапшы жүйе және оның құрамы: Оқу құралы. – Алматы, ҚазҰТУ, 2004 ж., 100 б.
28. Попов Э.В. Экспертные системы: Решение неформализованных задач в диалоге с ЭВМ. – Наука, 1987 ж. – 288 б.
29. Алексеева Е.Ф., Стефанюк В.Л., Экспертные системы – состояние и перспективы. // Изв.АН СССР Техн. кибернетика. 1984. – N 5. – б. 153-167.
30. Представление знаний в человеко-машинных и робото-технических системах. – М.: ВИНИТИ, 1984. Т-А. – 261 б.
31. Кондаков Н.И. Логический словарь справочник. – М.: Наука, 1976. – 720 б.
32. Построение экспертных систем. Под. редакцией Ф.Хейес Рот, Д.Уотерман, Д.Ленат – М.: Мир, 1987. – 441 б.
33. Д. Марселус. Программирование экспертных систем на Турбо Прологе. Пер. с анг. – М: Финансы и статистика, 1994. – 256 б.
34. Минский М., Пейперт С., Перцептроны. – М.: Мир, 1971. / Пер. с англ. 320 б.
35. Поспелов Д.А. Логико-лингвистические модели в системах управления. – М: Энергоиздат, 1981. – б. 231.
36. Уинстон П. Искусственный интеллект. – М.: Мир, 1980
37. Базы знаний и интеллектуальные системы /Т.А.Гаврилова, В.Ф.Хорошевский – СПб: Питер, 2000. – 384 б.
38. http: //ru.wukipedia.org/wiki/Логическое программирование.
39. Большакова Е.И., Мальковский М.Г., Пильщиков В.Н. Искусственный интеллект. Алгоритмы эвристического поиска (учебное пособие) – М.: Издательский отдел факультета ВМК МГУ, 2002. – 83 б.
40. Семенов М.Ю. Язык Лисп для персональных ЭВМ. – М.: Издательство МГУ, 1989.
41. Круглов В.В., Борисов В.В. Искусственные нейронные сети. Теория и практика. – М.: Горячая линия-Телеком, 2002. – 382 б.
42. Круг П.Г. Нейронные сети и Нейрокомпьютеры. Учебное пособне. –М.: Изд-во МЭИ, 202. – 176 б.
43. Комашинский В.И., Смирнов Д.А. Нейронные сети и их применение в системах управления и связи. – М.: Горячая линия-Телеком, 2003. – 94 б.
44. http://neurnews.iu4.bmstu.ru
45. http://neuronets.chat.ru
46. http://www.module.ru
47. http://www.scanti.ru
48. http://www.ti.com
49. http://www.trajan-software.demon.co.uk
50. http://neuropower.de/rus/books/index.html

МАЗМҰНЫ

Алғы сөз 3
Кіріспе 5
1 Логикалық программалау 9
1.1 Логикалық программалау тарихы 11
1.2 Пролог тілінің синтаксисі мен семантикасы 15
1.3 Тілдегі тізімдер мен арифметика 20
1.4 Пролог тіліндегі кесу 22
1.5 Пролог тіліндегі деректердің абстракты типтері 25
1.6 Дерекқормен жұмыс және рекурсивті іздеу 29
1.7 Пролог тіліндегі жаттығулар 39
Бақылау сұрақтары 54
2 Функционалдық программалау 55
2.1 Функционалдық программалау тарихы 55
2.2 Функционалдық программалау негіздері 58
2.3 Функционалдық программалау ерекшеліктері 65
2.4 Функционалдық программалау тілдері 75
2.5 Haskell тілінің мүмкіндіктері 77
2.6 Haskell тілінің негіздері 82
2.7 Лисп тілінің негіздері. Лисп тіліндегі рекурсия 91
2.8 Лисп тіліндегі есептеулер мен функционалдар 102
2.9 Функционалдық программалаудың қолданылуы 113
2.10 Лисп тіліндегі жаттығулар 116
Бақылау сұрақтары 134
3 Жасанды зерде жүйелері 136
3.1 Жасанды зерденің негізгі ұғымдары 137
3.2 Есептерді шешудің түрлі әдістері 141
3.3 Теоремаларды автоматты дәлелдеу әдістері 147
3.4 Предикаттарды есептеудегі іздеу әдістері 153
3.4.1 Күйлер кеңістігіндегі іздеу әдістері 156
3.4.2 Кеңістіктегі есептер жағдайы бойынша есептеу 160
3.4.3 Предикаттарды құру арқылы іздеу әдістері 168
3.5 Жасанды зерде есептерінің жаттығулары 176
3.5.1 «Соқыр әдістердің» жаттығулары 177
3.5.2 Редукция әдістеріне жаттығулар 188
3.5.3 Предикаттарды есептеу жаттығулары 193
Бақылау сұрақтары 199

4 Сарапшы жүйелер.........200
4.1 Білімдерді пайдалану үлгілері.........201
4.2 Сарапшы жүйені жобалау ерекшеліктері.........207
4.3 Продукциялық типтегі үлгі.........213
4.4 Фреймдік және семантикалық желі үлгілері.........218
4.5 СЖ құрудың кезеңдері мен құралдары.........225
4.6 СЖ құрудың жаттығулары.........227
4.6.1 Продукциялық үлгідегі СЖ құру.........229
4.6.2 Фреймдік үлгідегі СЖ құру.........239
Бақылау сұрақтары.........251
5 Үйрету жүйелері.........253
5.1 Жасанды нейрондық жүйелер.........253
5.1.1 Жасанды нейрон желілерінің тарихы.........253
5.1.2 Биологиялық нейрон.........258
5.1.3 Жасанды нейрон құрылымы.........260
5.1.4 Нейрон желісінің жүйеленуі.........266
5.1.5 Жасанды нейрон желілерін үйрету.........267
5.1.6 Жасанды нейрон желілерін қолдану аумағы.........269
5.1.7 Нейрон желілерінің программалық өнімдері.........272
5.2 Үйрету жүйелерінің әдістері.........273
5.2.1 Сұхбат арқылы үйрету.........274
5.2.2 Бейнені тану арқылы үйрету.........277
Бақылау сұрақтары.........283
6 Табиғи тілді түсіну.........285
6.1 Табиғи тілді түсіну мәселелері.........286
6.2 Таңбалы талдау.........288
6.3 Өту желілері негізіндегі талдауыштар.........293
6.4 Хомский иерархиясы.........297
6.5 Табиғи тілді түсіну жолдары.........300
6.6 Табиғи тілдегі сөз қорын құру.........301
Бақылау сұрақтары.........307
Қорытынды.........308
Қосымша.........312
Глоссарий.........315
Әдебиеттер тізімі.........326

Ахметова Майра

ФУНКЦИОНАЛДЫҚ-ЛОГИКАЛЫҚ ПРОГРАММАЛАУ ЖӘНЕ ЖАСАНДЫ ЗЕРДЕ ЖҮЙЕЛЕРІ

(Оқу құралы)

ISBN 978-601-7275-38-9

Корректор – **Омарова Сәуле Сыдықбекқызы**
Компьютерде беттеген – **Любовицкая Ольга**

Басуға 2011 жылы қол қойылды.
Форматы 60х84 1/16. Көлемі 20,75 баспа табақ.
Times гарнитурасы. Офсеттік басылым.
Тапсырыс № 28. Тиражы – 1500 дана.

«Бастау» баспасы (тел. 266-54-59, 266-54-58).
Мемлекеттік лицензия – № 0000036
ҚР Білім және ғылым министрлігі.
ҚР Ұлттық мемлекеттік кітап палатасының
халықаралық код беру туралы №155 –
978-601-281 сертификаты.
Алматы қаласы, Сейфуллин даңғылы, 458/460-95.

«Полиграфсервис» баспаханасында басылды (тел. 233-32-53),
Алматы қаласы, Зеленая көшесі, 13-а.

Жазбалар үшін

Жазбалар үшін