ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ
ՀԱՄԱԼՍԱՐԱՆ

Ա. Հ. ԿՈՍՏԱՆՅԱՆ

ԽՆԴԻՐՆԵՐԻ ԼՈՒԾՈՒՄ PROLOG ԼԵԶՎՈՎ

ՈՒՍՈՒՄՆԱՄԵԹՈԴԱԿԱՆ ՁԵՌՆԱՐԿ

ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

Ա. Հ. ԿՈՍՏԱՆՅԱՆ

ԽՆԴԻՐՆԵՐԻ ԼՈՒԾՈՒՄ **PROLOG** ԼԵԶՎՈՎ

Ուսումնամեթոդական ձեռնարկ

ԵՐԵՎԱՆ
ԵՊՀ ՀՐԱՏԱՐԱԿՉՈՒԹՅՈՒՆ
2016

ՀՏԴ 004(07)
ԳՄԴ 32.81գ7
Կ 756

Հրատարակության է երաշխավորել
ԵՊՀ ՏՏ կրթական և հետազոտական
կենտրոնի գիտական խորհուրդը

Խմբագիր՝ ԵՊՀ ՏՏ կրթական և հետազոտական կենտրոնի գիտական ղեկավար, ՀՀ ԳԱԱ ակադեմիկոս Ս. Կ. Շուքուրյան

Կ 756 Խնդիրների լուծում **Prolog** լեզվով: Ուսումնամեթոդական ձեռնարկ/**Ա. Հ. Կոստանյան**: -Եր., ԵՊՀ հրատ., 2016, 114 էջ:

Ձեռնարկում դիտարկվում է տրամաբանական ծրագրավորման Prolog լեզուն, որն այլընտրանք է պրոցեդուրային ծրագրավորման այնպիսի լեզուների, ինչպիսիք են **C++**-ը, **C#**-ը, **Java**-ն**, Pascal**-ը և այլն: Ներկայացվում են սիմվոլային տողերի, մատրիցների, գրաֆների, ավտոմատների և քերականությունների մշակման մի շարք խնդիրներ, որոնցում շեշտադրվում է խնդիրների լուծման ոչ պրոցեդուրային (դեկլարատիվ) մոտեցումը: **Prolog** լեզվի առավելությունները ընդգծելու համար հատուկ ուշադրություն է հատկացվում անդրադարձ տվյալների կառուցվածքների մշակման խնդիրներին: Ձեռնարկի վերջում բերվում են լեքսիկական և շարահյուսական վերլուծության խնդիրների ընդհանրացված լուծումներ:

Ձեռնարկը նախատեսված է ԵՊՀ տեղեկատվական համակարգերի ամբիոնի «Տեղեկատվական համակարգերի մշակում» և «Տեղեկատվական համակարգերի կառավարում» մագիստրոսական ծրագրերի ուսանողների, ինչպես նաև բոլոր նրանց համար, ովքեր հետաքրքրված են տրամաբանական ծրագրավորման կիրառություններով:

ՀՏԴ 004(07)
ԳՄԴ 32.81գ7

ISBN 978-5-8084-2127-1

ԲՈՎԱՆԴԱԿՈՒԹՅՈՒՆ

1. Ներածություն

Prolog-ը (***Pro***graming in ***Log***ic) տրամաբանական ծրագրավորման լեզու է: **Prolog** լեզվով ծրագրավորելիս խնդրի լուծումը նախ ներկայացվում է մուտքային և ելքային տվյալները միմյանց հետ կապող նպատակային պրեդիկատի միջոցով, այնուհետև խնդիր է դրվում կառուցելու փաստերից և արտածման կանոններից բաղկացած տրամաբանական համակարգ, որից ֆորմալ եղանակով հնարավոր լինի արտածել նպատակային պրեդիկատը:

Prolog լեզվում բացակայում են պրոցեդուրային լեզուներին (**C++, Java, C#** և այլն) բնորոշ տվյալների տիպերը և գործողությունների հաջորդականության ղեկավարումը: Փոխարենը օգտագործվում են պրեդիկատների սահմանման անդրադարձ (ռեկուրսիվ) կանոններ, ինչի շնորհիվ **Prolog**-ը վերածվում է դեկլարատիվ լեզվի: Այն ծրագրավորողին թելադրում է անդրադարձ մտածելակերպ և անդրադարձ ոճով խնդիրների լուծում: **Prolog** լեզուն հարմար է օգտագործել այնպիսի կիրառություններում, ինչպիսիք են անդրադարձ տվյալների կառուցվածքների մշակումը, սիմվոլային հաշվումները, փորձագիտական համակարգերի ստեղծումը, թեորեմների ապացույցը, տեքստերի թարգմանությունը:

1.1. Փաստեր, կանոններ, հարցումներ

1.1.1. Փաստեր

Փաստը **Prolog** լեզվում գրառվում է հետևյալ կերպ.

հարաբերություն (օբյեկտների ցուցակ).

որտեղ հարաբերությունը և օբյեկտները փոքրատառով սկսվող անուններ են: Նշենք, որ փաստը պարտադիր պետք է ավարտվի «կետ» նշանով: Օրինակ՝

likes(john, mary).

փաստով արձանագրվում է այն, որ **john**-ը հավանում է **mary**-ին: Եթե անհրաժեշտ է նշել, որ **mary**-ն նույնպես հավանում է **john**-ին, ապա պետք է նաև ավելացնել հետևյալ փաստը.

likes(mary, john).

Փաստերի հաջորդականությունը **Prolog** լեզվում կոչվում է պարզագույն տվյալների բազա: Օրինակ՝

```
likes(joe, fish).
likes(joe, mary).
likes(mary, book).
likes(joe, book).
```

1.1.2. Հարցումներ

Տվյալների բազային կարելի է դիմել հարցումներով: Հարցումը գրառվում է այնպես, ինչպես փաստն այն տարբերությամբ, որ հարցումից առաջ դրվում են **?-** նշանները: Օրինակ՝

```
?-likes(joe, money).
no

?-likes(mary, joe).
no

?-likes(mary, book).
yes
```

1.1.3. Փոփոխականներ

Դիցուք անհրաժեշտ է պարզել, թե ինչ է հավանում **joe**-ն: Դա կարելի էր անել հաջորդաբար հարցնելով՝ հավանում է արդյոք **joe**-ն այս կամ այն օբյեկտը: Փոխարենը կարելի է կազմել հետևյալ հարցումը.

```
?-likes(joe, X).
```

Այստեղ **X**-ը հանդես է գալիս փոփոխականի դերում, ինչը բխում է գրելաձևից. **Prolog** լեզվում մեծատառով սկսվող ցանկացած անուն դիտարկվում է որպես փոփոխական: Փոփոխականները կարող են լինել արժևորված կամ անորոշ: Նշենք, որ **Prolog** լեզվում փոփոխականներն ըստ տիպերի չեն դասակարգվում և ցանկացած օբյեկտ կարող է լինել ցանկացած փոփոխականի արժեք: Կասենք, որ փոփոխականը կոնկրետացված է, եթե տվյալ պահին նրան համապատասխանում է որևէ օբյեկտ: Այլապես կասենք, որ այն կոնկրետացված չէ:

Եթե հարցումը պարունակում է փոփոխականներ, ապա **Prolog** համակարգը հաջորդաբար դիտարկում է տվյալների բազայում գրառված փաստերը՝ որոնելով փոփոխականների այնպիսի կոնկրետացում, որի դեպքում հարցումը համընկնում է որևէ փաստի հետ: Ընդհանուր դեպքում, երբ տվյալների բազան պարունակում է կանոններ, փոփոխականների կոնկրետացումը կատարվում է հետդարձով որոնման եղանակով (**backtracking**):

Վերը դիտարկված օրինակի համար **X**-ն առաջին անգամ կոնկրետանում է՝ ընդունելով **fish** արժեքը: Դրանից հետո համակարգը սպասում է հետագա գործողությունների: Եթե հրահանգվում է շարունակել որոնումը, ապա **X**-ը կոնկրետանում է երկրորդ անգամ՝ ընդունելով **mary** արժեքը: Որոնումը կրկին շարունակելու դեպքում **X**-ը կոնկրետանում է երրորդ և վերջին անգամ՝ ընդունելով **book** արժեքը: Այսպիսով՝ ստանում ենք.

?-likes(joe, X).
X=fish?;
X=mary?;
X=book
yes

1.1.4. Կոնյունկցիաներ

Դիտարկենք տվյալների հետևյալ բազան:

likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).

Դիցուք անհրաժեշտ է պարզել՝ հավանում են արդյոք **john**-ը և **mary**-ն միմյանց, թե ոչ: Դա կարելի էր անել՝ նախ կատարելով.

?-likes(john, mary).

հարցումը և այնուհետև դրական պատասխան ստանալուց հետո՝

?-likes(mary, john).

հարցումը: Փոխարենը բավարար է կատարել հետևյալ համակցված հարցումը.

?-likes(john, mary), likes(mary, john).
no

Օգտագործելով փոփոխականներ կոնյունկցիաներում՝ կարող ենք կառուցել ավելի բարդ հարցումներ: Օրինակ՝

?-likes(john, X), likes(mary, X).
X=wine?;
no

1.1.5. Կանոններ

Կանոնները նշվում են հետևյալ կերպ.

վերնագիր:- մարմին.

(«**:-**» նշանակումը կարդացվում է «եթե»):

Օրինակ՝

likes(john, X):- likes(X, wine), likes(X, food).

կամ՝

likes(john, X):- woman(X), likes(X, wine).

Նշենք, որ կանոններում կարող են օգտագործվել ցանկացած թվով փոփոխականներ:

Ստորև բերված է տվյալների բազա, որը պարունակում է Անգլիայի Վիկտորյա թագուհու տոհմածառին առնչվող որոշ փաստեր.

man(albert).
man(edward).
woman(alice).
woman(victoria).
parents(edward, victoria, albert).
parents(alice, victoria, albert).

Սահմանենք **isSister** հարաբերությունը հետևյալ կանոնի միջոցով.

isSister(X, Y):- woman(X), parents(X, M, F), parents(Y, M, F).

Ավելացնենք այս կանոնը տվյալների բազային և ձևակերպենք հետևյալ հարցումը.

?-isSister(alice, X).
X=edward?;
X=alice
yes

X=alice անհեթեթ պատասխանը բացառելու համար **isSister** կանոնի աջ մասում պետք է պահանջել, որ **X**-ը և **Y**-ը հավասար չլինեն միմյանց:

Դիտարկենք մեկ այլ օրինակ.

```
thief(john).
likes(mary, food).
likes(mary, wine).
likes(john, X):-likes(X,wine).
can_steal(X, Y):-thief(X), likes(X, Y).
```

Նշենք, որ այս օրինակում **likes** պրեդիկատը օգտագործվում է միաժամանակ և՛ փաստերում, և՛ կանոններում:

```
?-can_steal(john, X).
X=mary?;
no
```

Վարժություն

Տրված են հետևյալ հարաբերությունները.

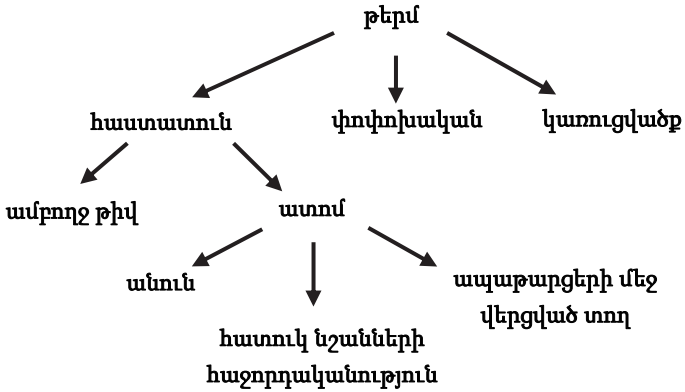
father(X, Y)	(**X**-ը **Y**-ի հայրն է)
mother(X, Y)	(**X**-ը **Y**-ի մայրն է)
man(X)	(**X**-ը տղամարդ է)
woman(X)	(**X**-ը կին է)
parent(X, Y)	(**X**-ը **Y**-ի ծնողն է)
different(X, Y)	(**X**-ը և Y-ը տարբեր են)

Հիմք ընդունելով այս հարաբերությունները՝ սահմանել հետևյալ նոր հարաբերությունները.

1. **mother1(X)** (**X**-ը մայր է)
2. **son(X)** (**X**-ը որդի է)
3. **sister(X, Y)** (**X**-ը **Y**-ի քույրն է)
4. **grandfather(X, Y)** (**X**-ը **Y**-ի պապն է)

2. Շարահյուսություն

Prolog ծրագրերը կառուցվում են թերմերից: Թերմը սահմանվում է հետևյալ կերպ.



Նկ. 2.1

2.1. Հաստատուններ, փոփոխականներ, կառուցվածքներ

2.1.1. Հաստատուններ

Գոյություն ունեն երկու տեսակի հաստատուններ՝ *ամբողջ թվեր* և *ատոմներ*:

Prolog լեզվում ամբողջ թվեր են համարվում տասական համակարգում ներկայացված և մեքենայական բառով սահմանափակված ամբողջ թվերը, որոնց նկատմամբ կիրառվում են թվաբանական և համեմատության հիմնական գործողությունները: Դրա հետ մեկտեղ **Prolog**-ը հնարավորություն է տալիս մոդելավորելու աշխատանքը մեքենայական բառով չսահմանափակված ամբողջ թվերի հետ թվաբանական և համեմատության գործողությունները իրականացնող ծրագրերի ստեղծման միջոցով:

Ատոմներ են՝

- տառերից, թվանշաններից և ընդգծման նշանից կառուցված, փոքրատառով սկսվող հաջորդականությունները: Օրինակ՝

alpha, aLIST01, a_LIST_01:

- Հատուկ նշաններից կառուցված հաջորդականությունները: Օրինակ՝

?-, :-, +, ->:

- Ապաթարցերի մեջ վերցված կամայական նշանների հաջորդականությունները: Օրինակ՝

’123’, ’Alpha’, ’1+2’ և այլն:

2.1.2. Փոփոխականներ

Փոփոխականը տառերից, թվանշաններից և ընդգծման նշանից կառուցված, մեծատառով կամ ընդգծման նշանով սկսվող հաջորդականություն է: Օրինակ՝

Alpha, WWW, _a_PLUS_b:

Միայն ընդգծման նշանից բաղկացած փոփոխականը կոչվում է *անանուն փոփոխական*: Ի տարբերություն սովորական փոփոխականի՝ անանուն փոփոխականի արժեքը չի կոնկրետանում: Օրինակ, եթե անհրաժեշտ է պարզել՝ հավանում է արդյոք որևէ մեկը **john**-ին, բավարար է գրել.

```
?- likes(_, john).
```

Համակարգը այս հարցմանը կտա **true** կամ **no** պատասխան: Ի տարբերություն դրան, եթե հարցումը ձևակերպվի

```
?- likes(X, john).
```

տեսքով, ապա համակարգը հաջորդաբար կարտահանի **X**-ի բոլոր հնարավոր արժեքները:

Եթե անանուն փոփոխականը հարցման մեջ հանդիպում է մեկ անգամից ավելի, ապա յուրաքանչյուր հանդիպում դիտարկվում է որպես առանձին փոփոխական: Օրինակ՝

```
?- likes(_, john), likes(_, mary).
```

հարցման պատասխանը կլինի **true** այն և միայն այն դեպքում, երբ կա երկու օբյեկտ, որոնցից մեկը հավանում է **john**-ին, իսկ մյուսը՝ **mary**-ին: Ի տարբերություն դրան, եթե հարցումը ձևակերպվի հետևյալ տեսքով՝

```
?- likes(X, john), likes(X, mary).
```

ապա պատասխանում կարտաբերվեն բոլոր այն օբյեկտները, որոնք միաժամանակ հավանում են և՛ **john**-ին, և՛ **mary**-ին:

2.1.3. Կառուցվածքներ

Կառուցվածքը առանձին բաղադրիչներից բաղկացած միասնական և անվանակոչված օբյեկտ է: Բաղադրիչների դերում կարող են հանդես գալ հաստատուններ, փոփոխականներ և այլ կառուցվածքներ: Կառուցվածքը հնարավորություն է տալիս հստակեցնելու օբյեկտի իմաստը՝ խմբավորելով տվյալները այս կամ այն եղանակով:

Օրինակ՝

```
has(john, programming_in_prolog).
```

փաստից պարզ չէ, թե **john**-ի ունեցած օբյեկտը գիրք է, այլ ոչ թե, օրինակ, ամսագիր: Ներմուծելով **book** անունով կառուցվածք, որն ունի գրքի հեղինակին և վերնագիրը նշող բաղադրիչներ, այս փաստը կարող ենք ներկայացնել ավելի հասկանալի տեսքով.

```
has(john, book(clocksin_melish, programming_in_prolog)).
```

book կառուցվածքի բաղադրիչների իմաստն ավելի թափանցիկ դարձնելու համար կարող ենք գրել.

has(john, book(author(clocksin_melish), title(programming_in_prolog)).

Փոփոխականներ պարունակող կառուցվածքները կարող են հանդիպել հարցումներում: Օրինակ՝

?-has(john, book(author(clocksin_melish), X).
?-has(john, book(author(clocksin_melish), _).
?-has(X, book(author(clocksin_melish), Y).

Ընդհանուր դեպքում կառուցվածքը սահմանվում է հետևյալ կերպ.

ֆունկտոր (բաղադրիչների ցուցակ),

որտեղ **ֆունկտոր**-ը ատոմ է, իսկ **բաղադրիչները**՝ կամայական թերմեր: Բաղադրիչների քանակը կոչվում է կառուցվածքի տեղայնություն **(arity)**: Ատոմը դիտարկվում է որպես **0** տեղանի կառուցվածք:

Նշենք, որ փաստը շարահյուսորեն կառուցվածք է, իսկ կանոնն ու հարցումը՝ հատուկ նշաններից և կառուցվածքներից բաղկացած հաջորդականություններ:

2.2. Թվաբանություն

2.2.1. Օպերատորներ

Prolog լեզվում թվաբանական արտահայտությունը սահմանվում է որպես թիվ, փոփոխական կամ այնպիսի կառուցվածք, որի ֆունկտորը գործողության նշան է, իսկ բաղադրիչները՝ թվաբանական արտահայտություններ: Միջածանցային ձևով գրառված թվաբանական արտահայտությունը կոչվում է օպերատորային գրառում: Օրինակ՝ **X+Y*Z** օպերատորային գրառմանը համապատասխանում է **+(X,*(Y,Z))** թվաբանական արտահայտությունը: Օպերատորային գրառման հիման վրա կառուցվածքը միարժեքորեն վերականգնելու հա-

մար անհրաժեշտ է օգտագործել գործողությունների հետևյալ բնութագրիչները.

- ներկայացում (նախածանցային, վերջածանցային կամ միջածանցային),
- առաջնայնություն,
- ասոցիատիվություն (ձախ կամ աջ):

Այսպես, օրինակ, **a+b/c** օպերատորային գրառումը վերծանվում է որպես **+(a,/(b,c))** թվաբանական արտահայտություն՝ գումարման համեմատ բաժանման գործողության ավելի բարձր առաջնայնությունից ելնելով: Իսկ **8/2/2** օպերատորային գրառումը՝ որպես **/(/(8,2),2)** թվաբանական արտահայտություն՝ բաժանման գործողության ձախ ասոցիատիվությունից ելնելով:

Թվաբանական արտահայտության (օպերատորային գրառման) արժեքը հաշվվում է միայն այն դեպքում, երբ այն հանդիպում է **is** նախասահմանված բինար պրեդիկատի երկրորդ արգումենտի տեղում: Այնպես որ միմյանցից տարբեր են համարվում օպերատորային հետևյալ գրառումները՝ **3+4, 4+3, 7**:

Թվաբանական գործողություններ կատարելու համար **Prolog**-ը տրամադրում է հետևյալ նախասահմանված օպերատորները.

+	գումարում,
-	հանում,
*****	բազմապատկում
//	ամբողջաթիվ բաժանում,
mod	մնացորդի որոշում,
/	իրական թվերի բաժանում:

2.2.2. Պրեդիկատներ

Ամբողջ թվերի համեմատության համար **Prolog**-ում սահմանված են հետևյալ պրեդիկատները.

X<Y	**X**-ը փոքր է **Y**-ից,
X>Y	**X**-ը մեծ է **Y**-ից,
X=<Y	**X**-ը փոքր է կամ հավասար **Y**-ին,
X>=Y	**X**-ը մեծ է կամ հավասար **Y**-ին:

Չի թույլատրվում այս պրեդիկատները օգտագործել փաստեր կառուցելու համար: Այսպես, օրինակ, **2>3** փաստը շարահյուսորեն ճիշտ է գրառված, սակայն **Prolog** համակարգը թույլ չի տա ավելացնել այն տվյալների բազային՝ տվյալների ամբողջականությունը չխախտելու համար:

Դիտարկենք հետևյալ օրինակը:

Դիցուք **governed(X, Y, Z)**-ը պնդում է, ըստ որի՝ **X** անձը ղեկավարել է տվյալ երկիրը՝ սկսած **Y** թվականից մինչև **Z** թվականը: Նշանակենք **president(X, Y**) պնդումը, ըստ որի՝ **X**-ը երկրի նախագահ է եղել **Y** թվականին: **president** պրեդիկատը կարող ենք արտահայտել **governed** պրեդիկատի միջոցով հետևյալ կերպ.

president(X, Y):- governed(X, A, B), Y>=A, Y=<B.

Prolog-ում սահմանված է միջածանցային տեսքով գրառվող **=** նույնականացման օպերատորը, որն ունի հետևյալ իմաստը: Դիցուք **X**-ը և **Y**-ը կամայական թերմեր են, որոնք կարող են պարունակել նաև չկոնկրետացված փոփոխականներ: **X=Y** պնդման մշակումը կատարվում է հետևյալ կանոնների համաձայն.

- **X=Y** պնդումը համարժեք է **Y=X** պնդմանը,
- եթե **X**-ը և **Y**-ը հաստատուններ են կամ հաստատուններով կոնկրետացված փոփոխականներ, ապա **X=Y** պնդումը բավարարվում է այն և միայն այն դեպքում, երբ **X**-ին և **Y**-ին համապատասխանող հաստատունները համընկնում են,
- եթե **X**-ը չկոնկրետացված փոփոխական է, ապա **X=Y** պնդումը նույնաբար բավարարվում է, և այն ապացուցելու արդյունքում **X**-ը նույնանում է **Y**-ի հետ: Օրինակ՝

?- X= book(clocksin_melish, programming_in_prolog).

պնդումը բավարարվում է, ինչը հավաստելու արդյունքում **X**-ը կոնկրետանում է՝ ընդունելով **book(clocksin_melish, programming_in_prolog)** արժեքը,

- եթե **X**-ը և **Y**-ը կառուցվածքներ կամ կառուցվածքներով կոնկրետացված փոփոխականներ են, ապա **X=Y** պնդումը բավարարվում է այն և միայն այն դեպքում, երբ.
  - համընկնում են **X**-ին և **Y**-ին համապատասխանող կառուցվածքների ֆունկտորները և արգումենտների քանակները,
  - բավարարվում են համապատասխան արգումենտների հավասարության մասին պնդումները:

  Օրինակ՝ հետևյալ պնդումը

  a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, g(H, i, j)))

  բավարարվում է, ինչը հավաստելու արդյունքում **B, C, E, F, H, J** փոփոխականները կոնկրետանում են՝ համապատասխանաբար ընդունելով **b, c, e, f, h, j** արժեքները:

Prolog լեզվում սահմանվում է նաև **'\='** պրեդիկատը, ընդ որում՝ **X\=Y** պնդումը բավարարվում է այն և միայն այն դեպքում, երբ չի բավարարվում **X=Y** պնդումը: Հարկ է նշել, որ, ի տարբերություն **X=Y** պնդմանը, **X\=Y** պնդումն ապացուցելիս փոփոխականների կոնկրետացում տեղի չի ունենում:

Հաշվումներ իրականացնելու համար **Prolog** լեզվում օգտագործվում է միջածանցային ձևով գրառվող **is** նախասահմանված բինար պրեդիկատը՝

X is Y,

որտեղ **Y**-ը թվաբանական արտահայտություն է: Պահանջվում է, որ **X is Y** պնդումն ապացուցելու պահին **Y**-ի բոլոր փոփոխականները կոնկրետացված լինեն և ընդունեն թվային արժեքներ:

X is Y պնդումն ապացուցվում է հետևյալ եղանակով.

- հաշվվում է **Y** արտահայտության արժեքը, դիցուք՝ **A**,
- եթե **X**-ը չկոնկրետացված փոփոխական է, ապա **X is Y** պնդումը հավաստի է համարվում, ինչն ապացուցելու արդյունքում **X**-ը կոնկրետանում է՝ ընդունելով **A** արժեքը,
- հակառակ դեպքում **X is Y** պնդումը բավարարվում է այն և միայն այն դեպքում, երբ **X**-ը **A** հաստատուն կամ **A** հաստատունով կոնկրետացված փոփոխական է:

Դիտարկենք հետևյալ օրինակը:

Դիցուք **population(X, Y)**-ը պնդում է, ըստ որի՝ **X** երկրի բնակչությունը կազմում է **Y** մարդ, իսկ **area(X, Y)**-ը՝ պնդում, ըստ որի՝ **X** երկրի տարածքը հավասար է **Y** քառակուսի կիլոմետրի: Նշանակենք **density(X, Y)** պնդումը, ըստ որի՝ **X** երկրի բնակչության խտությունը կազմում է **Y** մարդ մեկ քառակուսի կիլոմետրի վրա: **density** պրեդիկատը կարող ենք արտահայտել **population** և **area** պրեդիկատների միջոցով հետևյալ կերպ.

density(X, Y):- population(X, P), area(X, A), Y is P/A.

Վարժություններ

1. Ստորև բերված օբյեկտներից նշել շարահյուսորեն ճիշտ կազմված կառուցվածներն ու դրանց տեսակը՝ *ատոմ*, *թիվ*, *փոփոխական* կամ *կառուցվածք*:

a) Diana
b) diana
c) 'Diana'
d) _diana
e) 'Diana goes south'
f) goes(diana, south)
g) 45
h) 5(X, Y>
i) -(north, west)
j) three(Black(Cats)).

2. Տրված է հետևյալ ծրագիրը.

```
f(1, one).
f(s(1), two).
f(s(s(1)), three).
f(s(s(s(X))), N) :- f(X, N).
```

Ցույց տալ ստորև բերված հարցումների մշակման ընթացքը.

a) ?- f(s (1), A).

b) ?- f(s(s(1)), two).

c) ?- f(s(s(s(s(s(s(1)))))), C).

d) ?- f(D, three).

3. Անդրադարձում և հատում: Թվային խնդիրների լուծում

3.1. Անդրադարձում

Անդրադարձումը առաջանում է այն դեպքում, երբ տվյալների բազան պարունակում է *անդրադարձ կանոններ*, այսինքն՝ այնպիսի կանոններ, որոնց աջ մասում հանդիպում է ձախ մասում օգտագործված պրեդիկատը: Անդրադարձումը **Prolog** լեզվում ունի առանձնահատուկ նշանակություն. այն փոխարինում է պրոցեդուրային ծրագրավորման լեզուներում օգտագործվող ցիկլի հրահանգին: Դիտարկենք անդրադարձման օգտագործման մի շարք օրինակներ:

3.1.1. Ֆակտորիալի հաշվում

Ֆակտորիալ ֆունկցիայի անդրադարձ սահմանումն է.

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ (n-1)! \cdot n, & \text{if } n > 0 \end{cases}$$

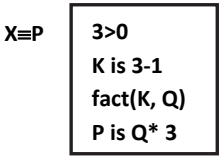
Նշանակենք **fact(N, P)** պնդումը, ըստ որի՝ **P**-ն **N** թվի ֆակտորիալն է: Հենվելով ֆակտորիալ ֆունկցիայի անդրադարձ սահմանման վրա՝ **fact(N, P)** պրեդիկատը (հարաբերությունը) ներկայացնենք հետևյալ կերպ.

```
fact(0, 1).
fact(N, P):-N>0, K is N-1, fact(K, Q), P is Q*N.
```

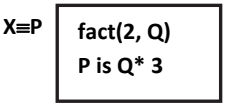
Նկ. 3.1-ում լուսաբանված է **Prolog**-ի ինտերպրետատորի աշխատանքը հետևյալ հարցման դեպքում.

```
?-fact(3, X).
```

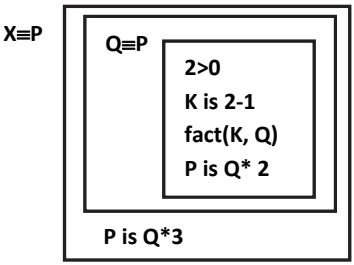
Քայլ 1.



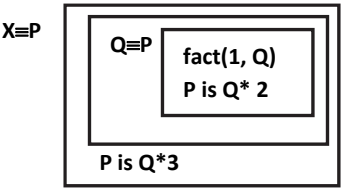
Քայլ 2.



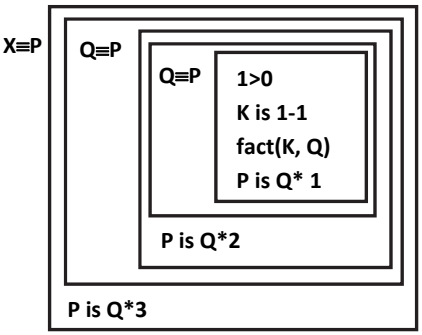
Քայլ 3.



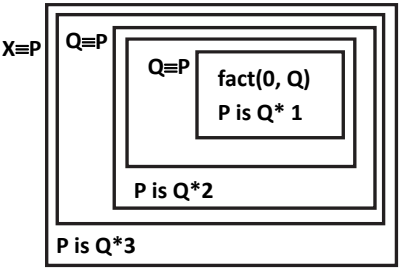
Քայլ 4.

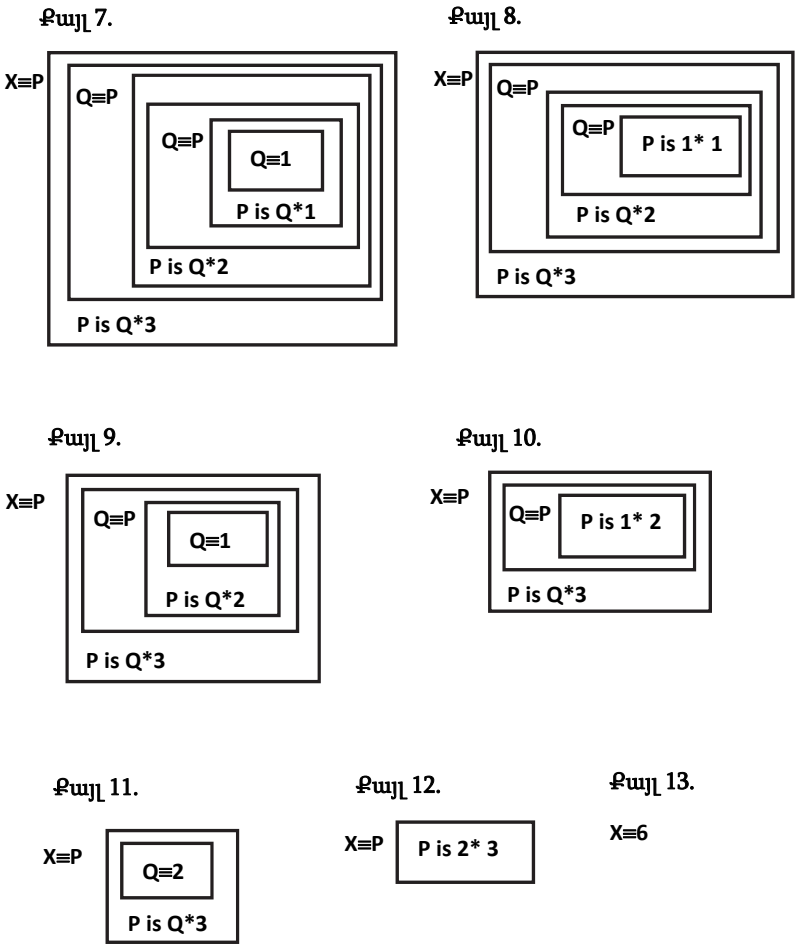


Քայլ 5.



Քայլ 6.





Նկ. 3.1

3.1.2. Ամենամեծ ընդհանուր բաժանարարի հաշվում

Երկու բնական թվերի ամենամեծ ընդհանուր բաժանարարը կարելի է հաշվել էվկլիդեսի ալգորիթմով, որի անդրադարձ ներկայացումն է.

$$gcd(x, y) = \begin{cases} x, & \text{if } x = y \\ gcd(x - y, y), & \text{if } x > y \\ gcd(x, y - x), & \text{if } y > x \end{cases}$$

Նշանակենք **gcd(X, Y, D)** պնդումը, ըստ որի՝ **D**-ն **X** և **Y** թվերի ամենամեծ ընդհանուր բաժանարարն է: Վերը բերված անդրադարձ նկարագրին համապատասխանում է **gcd** պրեդիկատի հետևյալ սահմանումը.

```
gcd(X, X, X).
gcd(X, Y, D):- X>Y, X1 is X-Y, gcd(X1, Y, D).
gcd(X, Y, D):- Y>X, Y1 is Y-X, gcd(X, Y1, D).
```

3.1.3. Աստիճանի հաշվում

Բնական թվի ամբողջ ոչ բացասական աստիճանը կարելի է հաշվել հետևյալ անդրադարձ եղանակով.

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot x^{n-1}, & \text{if } n > 0 \end{cases}$$

Նշանակենք **power(X, N, Y)** պնդումը, ըստ որի՝ **X** բնական թվի **N**-րդ աստիճանն է **Y**-ը: Վերը բերված բանաձևին համապատասխանում է **power** պրեդիկատի հետևյալ սահմանումը.

```
power(_, 0, 1).
power(X, N, Y):- N>0, N1 is N-1, power(X, N1, Y1), Y is Y1*X.
```

Նկատենք, որ աստիճանի հաշվումը ավելի արդյունավետ կլինի հետևյալ բանաձևի օգտագործման դեպքում.

$$x^n = \begin{cases} 1, & \text{if } = 0 \\ x^{n/2} \cdot x^{n/2}, & \text{if } n > 0 \text{ and } n \text{ is even} \\ x^{n/2} \cdot x^{n/2} \cdot x, & \text{if } n > 0 \text{ and } n \text{ is odd} \end{cases}$$

Հիմք ընդունելով այս բանաձևը՝ **power** պրեդիկատը սահմանենք հետևյալ կերպ.

```
power(_, 0, 1).
power(X, N, Y):-  N>0, 0 is N mod 2, M is N//2,
                                power(X, M, Z), Y is Z*Z.
power(X, N, Y):-  N>0, 1 is N mod 2, M is N//2,
                                power(X, M, Z), Y is Z*Z*X.
```

3.1.4. Ֆիբոնաչիի շարքի N-րդ անդամի հաշվում

Ֆիբոնաչիի շարքի **N**-րդ անդամը սահմանվում է հետևյալ անդրադարձ հավասարման միջոցով.

$$f_n = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-2} + f_{n-1}, & \text{if } n \geq 2 \end{cases}$$

Նշանակենք **fib(N, X)** պնդումը, ըստ որի Ֆիբոնաչիի շարքի **N**-րդ անդամն է **X**-ը: **fib** պրեդիկատի սահմանումն է.

```
fib(0, 1).
fib(1, 1).
fib(N, X) :-  N>=2, N2 is N-2, N1 is N-1,
                    fib(N2, X2), fib(N1, X1), X is X2+X1.
```

Նկատենք, որ այս ծրագրով Ֆիբոնաչիի շարքի միևնույն անդամը հաշվվում է բազմաթիվ անգամ: Այսպես, օրինակ, **fib(5, X)** հարցումը ծնում է **fib(3, X)** և **fib(4, X)** հարցումները, իսկ վերջինս՝ կրկին **fib(3, X)** հարցումը: Ավելի արդյունավետ ծրագիր ստանալու համար սահմա-

նենք **fib1(N, X1, X)** օժանդակ պնդում, որը բավարարվում է այն և միայն այն դեպքում, երբ **N>0**, իսկ **X1**-ը և **X**-ը համապատասխանաբար Ֆիբոնաչիի շարքի **(N-1)**-րդ և **N**-րդ անդամներն են:

fib1 պրեդիկատի սահմանումն է.

```
fib1(1, 1, 1).
fib1(N, X1, X):-  N>1, N1 is N-1,
                        fib1(N1, X2, X1), X is X2+X1.
```

Հիմք ընդունելով **fib1** պրեդիկատը՝ **fib** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
fib(0, 1).
fib(N, X):-N>0, fib1(N, _, X).
```

3.2. Հատում

3.2.1. Ընդհանուր դրույթներ

Prolog-ի ինտերպրետատորը այս կամ այն նպատակային պընդումն ապացուցելիս հաջորդաբար դիտարկում է բոլոր հնարավոր տարբերակները՝ կիրառելով *վերադարձով որոնման* **(backtracking)** մեթոդը: Սակայն բոլոր տարբերակների կուրորեն դիտարկումը ծրագիրը կարող է դարձնել ոչ արդյունավետ:

Դիտարկենք, օրինակ, հետևյալ կանոնը.

```
fatherfather(X, Y):-  father(Z, Y), father(X, Z).
```

որով հավաստվում է, որ **X**–ը **Y**–ի հոր հայրն է, եթե որևէ **Z**–ը **Y** –ի հայրն է, իսկ **X** –ը՝ այդ **Z**–ի:

Պարզ է, որ առաջին կամ երկրորդ նպատակային պնդումն ապացուցելուց հետո վերադարձը դեպի ետ և նոր տարբերակների որոնումը անիմաստ է (կենսաբանական այլ հայրեր չեն կարող լինել): Նման իրավիճակներում ավելորդ տարբերակների դիտարկումը կարելի է

բացառել՝ օգտագործելով *հատում* կոչվող **0** տեղանի նախասահմանված պրեդիկատը, որը նշանակվում է **!**-ով: **!** պրեդիկատը համադրվում է ցանկացած տվյալների բազայի հետ, այն է՝ **!** պնդումը նույնաբար ճիշտ է համարվում: Միևնույն ժամանակ **!** պնդման ապացույցը առաջացնում է կողմնակի էֆեկտ. **Prolog**-ի ինտերպրետատորն այլ տարբերակներ որոնելու համար նշված կետում չի կիրառում վերադարձի մեխանիզմը:

Դիտարկված օրինակում ծրագրի աշխատանքը կարող ենք դարձնել ավելի արդյունավետ, եթե վերը նշված կանոնը արտագրենք հետևյալ տեսքով.

```
fatherfather(X, Y):-  father(Z, Y), !, father(X, Z), !.
```

Այսպիսով՝ հատումը հնարավորություն է տալիս.

- արագացնելու ծրագրի կատարումը՝ բացառելով անիմաստ տարբերակների դիտարկումը,
- խնայելու հիշողություն՝ նվազեցնելով վերադարձի հնարավոր կետերի քանակը:

Հատման մեխանիզմը նպատակահարմար է կիրառել անվերջ անդրադարձումը վերացնելու, ինչպես նաև միմյանց բացառող տարբերակների դիտարկումը արդյունավետ դարձնելու համար:

3.2.2. Անվերջ անդրադարձման բացառում

Դիտարկենք ֆակտորիալի հաշվման հետևյալ ծրագիրը՝

```
fact(0, 1).
fact(N, P):-  K is N-1, fact(K, Q), P is Q*N.
```

Եթե դիմենք այս ծրագրին

```
?-  fact(0, X).
```

հարցումով, ապա համակարգը կտա

```
X=1?
```

պատասխանը: Սակայն, եթե մենք պահանջենք շարունակել որոնումը, ապա կանցնենք **fact(-1, X)** պնդման ապացույցին, այնուհետև՝ **fact(-2, X)** պնդման ապացույցին, և այսպես շարունակ: Նկատենք, որ եթե ծրագրի տողերը փոխանակվեն տեղերով, ապա անվերջ անդրադարձում կառաջանա ցանկացած դեպքում:

Բերված ծրագրում անվերջ անդրադարձումը վերացնելու համար բավարար է հատման պրեդիկատը աջ մասում պարունակող կանոնի վերածել առաջին փաստը.

```
fact(0, 1):- !.
fact(N, P):- K is N-1, fact(K, Q), P is Q*N.
```

3.2.3. Միմյանց բացառող պնդումների ծրագրավորում

Դիտարկենք հետևյալ ֆունկցիան՝

$$f(x)=\begin{cases} -1, & \text{if } x < -1 \\ 0, & \text{if } -1 \le x \le 1 \\ 1, & \text{if } x > 1 \end{cases}$$

Եթե այս սահմանումը տառացիորեն ներկայացնենք **Prolog** լեզվով, ապա կստանանք.

```
f(X, -1) :- X<-1.
f(X,  0) :- -1=<X, X=<1.
f(X, 1)  :- X>1.
```

Բերված ծրագիրը կդառնա ավելի արդյունավետ, եթե օգտագործվի հատման մեխանիզմը.

```
f(X, -1) :- X<-1, !.
f(X,  0) :- X=<1, !.
f(X, 1).
```

Այս ծրագրում չեն դիտարկվում ավելորդ տարբերակներ, և չեն կատարվում ավելորդ ստուգումներ: Նկատենք, որ **!** պրեդիկատն այստեղ էական նշանակություն ունի. եթե գրենք.

```
f(X, -1) :-  X<-1.
f(X,  0) :-  X=<1.
f(X, 1).
```

ապա կառաջանա **Prolog** համակարգի հետ գործակցության հետևյալ սցենարը.

```
?- f(-10, Y).
Y=-1?;
Y=0 ?;
Y=1
yes
```

3.2.4. Հատման հատկությունները

Հատման գործողությունը ավելի բարդ է դարձնում **Prolog** ծրագրի հիմքում ընկած տրամաբանությունը:

Դիտարկենք հետևյալ օրինակը.

```
P:- A, B.
P:-C.
```

որտեղ **A**-ն, **B**-ն, **C**-ն և **P**-ն տրամաբանական պնդումներ են:

Այս ծրագիրը ներկայացնում է մի պնդում **P**-ի մասին, որն արտահայտվում է հետևյալ բանաձևի միջոցով.

$$P \Leftrightarrow (A \,\&\, B) \vee C,$$

ընդ որում՝ այս բանաձևը արդարացի կլինի նաև ծրագրի տողերը տեղափոխելու դեպքում:

Այժմ դիտարկենք հետևյալ համարժեքությունները.

P:- A, !, B.
P:- C.

համարժեք է $P \Leftrightarrow (A \& B) \vee (\neg A \& C)$

P:-C.
P:- A, !, B.

համարժեք է $P \Leftrightarrow C \vee (A \& B)$

Բերված օրինակներից հետևում է, որ կրճատման գործողությունը օգտագործելու դեպքում ծրագրի տրամաբանությունը ավելի բարդ է դառնում, և ծրագիրը զգայուն է լինում փաստերի և կանոնների տեղափոխության նկատմամբ: Սա այն գինն է, որն անհրաժեշտ է վճարել բարձր արդյունավետություն ապահովելու համար:

3.2.5. fail և true նպատակային պնդումներ

Երբեմն նպատակահարմար է օգտագործել **fail** և **true 0** տեղանի ներդրված պրեդիկատները, որոնց հետ կապված է հետևյալ իմաստը. **fail** պրեդիկատի ապացույցը մշտապես անհաջողությամբ է ավարտվում, իսկ **true** պրեդիկատի ապացույցը, ընդհակառակը, մշտապես հաջողությամբ:

Օրինակ՝ **P** պնդման ժխտումը ներկայացնող **Q** պրեդիկատը կարող ենք սահմանել հետևյալ կերպ.

Q:-P, !, fail.
Q:-true.

3.2.6. Ամփոփում

Կիրառենք հատման գործողությունը՝ այս բաժնում դիտարկված ծրագրերն ավելի արդյունավետ դարձնելու համար:

Ֆակտորիալի հաշվում

```
fact(0, 1) :- !.
fact(N, P) :- K is N-1, fact(K, Q), P is Q*N.
```

Ամենամեծ ընդհանուր բաժանարարի հաշվում

```
gcd(X, X, X) :- !.
gcd(X, Y, D) :- X>Y, X1 is X-Y, gcd(X1, Y, D), !.
gcd(X, Y, D) :- Y1 is Y-X, gcd(X, Y1, D).
```

Աստիճանի հաշվում

```
power(_, 0, 1) :- !.
power(X, N, Y) :- 0 is N mod 2, M is N//2, power(X, M, Z), Y is Z*Z, !.
power(X, N, Y) :- M is N//2, power(X, M, Z), Y is Z*Z*X.
```

Ֆիբոնաչիի շարքի N-րդ անդամի հաշվում

```
fib1(1, 1, 1) :- !.
fib1(N, X1, X) :- N1 is N-1, fib1(N1, X2, X1), X is X2+X1.

fib(0, 1) :- !.
fib(N, X) :- fib1(N, _, X).
```

Խնդիրներ

1. Որոշել տրված բնական թվի տասական թվանշանների գումարը:
2. Որոշել տրված բնական թիվը պալինդրո՞մ է, թե՞ ոչ (այն է՝ ճի՞շտ է արդյոք, որ համընկնում են այդ թվի առաջին ու վերջին, երկրորդ ու նախավերջին և այդպես հաջորդաբար թվանշանները):

4. Ցուցակներ

4.1. Տիպերը Prolog լեզվում

Prolog լեզվում բացակայում է տվյալների և փոփոխականների դասակարգումն ըստ տիպերի, սակայն հնարավորություն կա մոդելավորելու տիպը որպես թերմերի բազմություն՝ այն մշակող գործողությունների հետ մեկտեղ: Անդրադարձ կանոնների օգտագործումը հնարավորություն է տալիս սահմանելու տվյալների այնպիսի վերացական տիպեր, ինչպիսիք են ցուցակը, բազմանդամը, բինար ծառը և այլն: Նման տիպերի մշակումն իրականացվում է տիպը սահմանող անդրադարձ կանոնների ուղեկցմամբ: **Prolog** լեզվում կան ներդրված վերացական տիպեր՝ նախասահմանված գործողություններով: Դրանցից կարևորագույնն է *ցուցակը*:

4.2. Ցուցակ

4.2.1. Սահմանում

Ցուցակը ներդրված տիպ է, որի արժեքների բազմությունն է ստորև սահմանված **list** պրեդիկատին բավարարող թերմերի բազմությունը.

```
list([]):- !.
list(.(_, T)):- list(T).
```

Նշված տիպի արժեքները (թերմերը) կոչվում են ցուցակներ: Ցուցակին համապատասխանում է տարրերի հաջորդականություն, որը սահմանվում է հետևյալ անդրադարձ եղանակով.

- **[]** թերմին համապատասխանում է դատարկ տարրերի հաջորդականությունը,
- **.(H, L)** թերմին համապատասխանում է տարրերի այնպիսի հաջորդականություն, որի առաջին տարրն է **H**-ը, իսկ մնացած

տարրերի հաջորդականությունն է **L**-ին համապատասխանող տարրերի հաջորդականությունը:

Ցուցակը (թերմը) ստորև կնույնացնենք դրան համապատասխանող տարրերի հաջորդականության հետ: Ցուցակի օրինակներ են.

[]	դատարկ ցուցակ,
.(alpha, [])	**alpha** տարրից բաղկացած ցուցակ,
.(a, .(b(10), .(c, [])))	**a, b(10), c** տարրերից բաղկացած ցուցակ,
.([], .(a, []))	դատարկ և **a** տարրը պարունակող ցուցակներից բաղկացած ցուցակ:

Ցուցակների համար սահմանված են մի շարք պարզեցված գրելաձևեր: Նշենք դրանցից մի քանիսը.

- **.(E1, .(E2, … .(En, []) …))** ցուցակը թույլատրվում է գրառել

  [E1, E2, …, En]

  տեսքով,
- **.(H, T)** ցուցակը թույլատրվում է գրառել

  [H | T]

  տեսքով: Ըստ ընդունված տերմինաբանության՝ **H**-ը կոչվում է ցուցակի *գլուխ (**head**)*, իսկ **T**-ն՝ ցուցակի *պոչ (**tail**)*,
- **.(H1, .(H2, …, (Hk, T) …))** ցուցակը թույլատրվում է գրառել

  [H1, H2, …, Hk | T]

  տեսքով: Այստեղ **H1**-ը, **H2**-ը, **…, Hk**-ն ներկայացնում են ցուցակի առաջին **k** տարրերը, իսկ **T**-ն՝ ցուցակի պոչը:

Հետևյալ գրառումները համարժեք են միմյանց.

.(a, [])	⇔	**[a| []]**	⇔	**[a]**
.(a, .(b, []))	⇔	**[a| [b]]**	⇔	**[a,b]**
.(a, .(b, L))	⇔	**[a| [b|L]]**	⇔	**[a,b| L]**

4.2.2. Գործողություններ

Ստորև դիտարկենք ցուցակների մշակման մի շարք գործողություններ, որոնց մի մասը նախասահմանված է **Prolog** լեզվում:

Երկարության որոշում: Ցուցակի երկարությունը (տարրերի քանակը) սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ ցուցակի երկարությունը **0** է,
- ոչ դատարկ ցուցակի երկարությունը մեկով ավելի է ցուցակի պոչի երկարությունից:

Նշանակենք **length(L, N)** պնդումը, ըստ որի՝ **L** ցուցակի երկարությունը **N** է: Հիմք ընդունելով ցուցակի երկարության որոշման վերը բերված անդրադարձ կանոնը՝ **length** պրեդիկատը սահմանենք հետևյալ կերպ.

```
length([], 0):-!.
length([_|L], N):-length(L, N1), N is N1+1.
```

Պատկանելիություն: Տարրի պատկանելիությունը ցուցակին սահմանենք հետևյալ անդրադարձ եղանակով.

- տարրը պատկանում է այդ տարրով սկսվող ցուցակին,
- տարրը պատկանում է իրենից տարբերվող տարրով սկսվող ցուցակին, եթե այն պատկանում է այդ ցուցակի պոչին:

Նշանակենք **element(X, L)** պնդումը, ըստ որի՝ **X**-ը **L** ցուցակի տարրն է: Վերը բերված կանոնի համաձայն՝ **element** պրեդիկատը սահմանենք հետևյալ կերպ.

```
element(X, [X|_]):-!.
element(X, [_|L]):-element(X, L).
```

Նախածանցի որոշում: Տրված երկու ցուցակների համար մեկը մյուսի նախածանց լինելը սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ ցուցակը ցանկացած ցուցակի նախածանց է,

- ոչ դատարկ ցուցակը այլ (ոչ դատարկ) ցուցակի նախածանց է, եթե այդ ցուցակների առաջին տարրերը համընկնում են, և առաջին ցուցակի պոչը երկրորդ ցուցակի պոչի նախածանց է:

Նշանակենք **prefix(L1, L)** պնդումը, ըստ որի՝ **L1**-ցուցակը **L** ցուցակի նախածանց է: Վերը բերված կանոնի համաձայն՝ **prefix** պրեդիկատը սահմանենք հետևյալ կերպ.

```
prefix([], _):-!.
prefix([X|L1], [X|L]):-prefix(L1, L).
```

Վերջածանցի որոշում: Տրված երկու ցուցակների համար մեկը մյուսի վերջածանց լինելը սահմանենք հետևյալ անդրադարձ եղանակով.

- յուրաքանչյուր ցուցակ իր իսկ վերջացանց է,
- միմյանցից տարբեր երկու ցուցակներից մեկը մյուսի վերջածանց է, եթե առաջին ցուցակը երկրորդ ցուցակի պոչի վերջածանց է:

Նշանակենք **suffix(L1, L)** պնդումը, ըստ որի՝ **L1** ցուցակը **L** ցուցակի վերջածանց է: Վերը բերված կանոնի համաձայն՝ **suffix** պրեդիկատը սահմանենք հետևյալ կերպ.

```
suffix(L, L):-!.
suffix(L1, [_|L]):-suffix(L1, L).
```

Ենթացուցակի որոշում: Տրված երկու ցուցակների համար մեկը մյուսի ենթացուցակ լինելը սահմանենք հետևյալ անդրադարձ եղանակով.

- եթե առաջին ցուցակը երկրորդի նախածանց է, ապա այն երկրորդ ցուցակի ենթացուցակ է,
- հակառակ դեպքում առաջին ցուցակը երկրորդի նախածանց է, եթե այն երկրորդ ցուցակի պոչի ենթացուցակ է:

Նշանակենք **sublist(L1, L)** պնդումը, ըստ որի՝ **L1** ցուցակը **L** ցուցակի ենթացուցակ է: Վերը բերված կանոնի համաձայն՝ **sublist** պրեդիկատը սահմանենք հետևյալ կերպ.

```
sublist(L1, L):- prefix(L1, L),!.
sublist(L1, [_|L]):- sublist(L1, L).
```

Միակցում: Երկու ցուցակների միակցումը սահմանենք հետևյալ անդրադարձ եղանակով.

- եթե առաջին ցուցակը դատարկ է, ապա միակցման արդյունքն է երկրորդ ցուցակը,
- հակառակ դեպքում միակցման արդյունքն է այն ցուցակը, որի գլուխն է առաջին ցուցակի գլուխը, պոչն է առաջին ցուցակի պոչի և երկրորդ ցուցակի միակցումը:

Նշանակենք **append(L1, L2, L)** պնդումը, ըստ որի՝ **L** ցուցակը **L1** և **L2** ցուցակների միակցումն է: **append** պրեդիկատի սահմանումն է.

```
append([], L, L):-!.
append([X|L1], L2, [X|L]):-append(L1, L2, L).
```

Շրջում: Ցուցակի շրջումը սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ ցուցակի շրջումն է դատարկ ցուցակը,
- ոչ դատարկ ցուցակի շրջումն է շրջված պոչի միակցումը գլխից բաղկացած ցուցակի հետ:

Նշանակենք **reverse(L1, L2)** պնդումը, ըստ որի՝ **L2**-ցուցակը **L1** ցուցակի շրջումն է: **reverse** պրեդիկատի սահմանումն է.

```
reverse([], []):-!.
reverse([X|L1], L):-reverse(L1, L2), append(L2, [X], L).
```

Միաձուլում: Երկու կարգավորված ցուցակների միաձուլումը սահմանենք հետևյալ անդրադարձ եղանակով.

- եթե միաձուլվող ցուցակներից որևէ մեկը դատարկ է, ապա միաձուլման արդյունքն է մյուս ցուցակը,
- հակառակ դեպքում միաձուլման արդյունքն է այն ցուցակը, որի գլուխն է ցուցակների գլուխներից փոքրագույնը, պոչն է փոքրագույն գլխով ցուցակի պոչի միաձուլումը մյուս ցուցակի հետ:

Նշանակենք **merge(L1, L2, L)** պնդումը, ըստ որի՝ **L** ցուցակը **L1** և **L2** կարգավորված ցուցակների միաձուլումն է: **merge** պրեդիկատի սահմանումն է.

```
merge([], L, L):- !.
merge(L, [], L):- !.
merge([X|L1], [Y|L2], [X|L]):- X=<Y, merge(L1, [Y|L2], L), !.
merge(L1, [Y|L2], [Y|L]):- merge(L1, L2, L).
```

Խնդիրներ

1. Սահմանենք *հիերարխիկ ամբողջաթիվ ցուցակ* տվյալների վերացական տիպ հետևյալ կերպ.
   - ամբողջ թիվը հիերարխիկ ամբողջաթիվ ցուցակ է,
   - հիերարխիկ ամբողջաթիվ ցուցակներից կազմված (հնարավոր է դատարկ) ցուցակը հիերարխիկ ամբողջաթիվ ցուցակ է:

   Հիերարխիկ ամբողջաթիվ ցուցակներ են, օրինակ, հետևյալ ցուցակները՝

   1, [], [[2]], [[1,3], [[],[4]], 2]:

   Որոշել առնվազն մեկ թիվ պարունակող հիերարխիկ ամբողջաթիվ ցուցակի մաքսիմալ տարրը: Օրինակ՝

   [[1, 3], [[], [4]], 2]

   ցուցակի մաքսիմալ տարրն է **4**-ը:

   Կառուցել այս խնդրի լուծման ծրագրեր **Prolog** և **C++** լեզուներով:
2. Կառուցել տրված **N** բնական թիվը չգերազանցող պարզ թվերի ցուցակը:

   Օգտագործել Էրատոսֆենի ալգորիթմը, ըստ որի՝ նախ ստեղծվում է **N**-ը չգերազանցող կենտ թվերի ցուցակը, այնուհետև յուրաքանչյուր հերթական թվի համար ցուցակի մնացած մասից հեռացվում են այդ թվի վրա առանց մնացորդի բաժանվող թվերը: Ի վերջո, ստացված ցուցակի սկզբում ավելացվում է **2**-ը, եթե **N≥2**:

5. Տեսակավորման ալգորիթմներ

Ստորև կառուցենք տեսակավորման մի քանի հանրահայտ ալգորիթմների ծրագրերը **Prolog** լեզվով՝ ենթադրելով, որ տեսակավորման ենթակա հաջորդականությունը ներկայացված է ցուցակի տեսքով:

5.1. Պղպջակների մեթոդ

Տեսակավորման պղպջակների մեթոդը սահմանենք հետևյալ անդրադարձ եղանակով.

- եթե ցուցակը չի պարունակում չկարգավորված հարևան տարրեր, ապա այն տեսակավորված է,
- հակառակ դեպքում ցուցակի տեսակավորման արդյունքն է այն ցուցակը, որը ստացվում է առաջին հանդիպած չկարգավորված հարևան տարրերը միմյանց հետ տեղափոխելու և ստացված ցուցակը պղպջակների մեթոդով տեսակավորելու արդյունքում:

Նշանակենք **makeSwap(L1, L)** պնդումը, ըստ որի՝ **L1** առնվազն երկու տարր պարունակող ցուցակում կան չկարգավորված հարևան տարրեր, և **L** ցուցակը ստացվում է **L1** ցուցակից առաջին հանդիպած այդպիսի տարրերը միմյանց հետ տեղափոխելու արդյունքում: **makeSwap** պրեդիկատը սահմանենք հետևյալ կերպ.

```
makeSwap([X, Y| L], [Y, X|L]):-X>Y,!.
makeSwap([X, Y| L], [X| L1]):-makeSwap([Y|L], L1).
```

Հիմք ընդունելով **makeSwap** պրեդիկատը՝ պղպջակների մեթոդով տեսակավորման ալգորիթմը ներկայացնենք հետևյալ կերպ.

```
bubbleSort(L,S):-makeSwap(L, L1), bubbleSort(L1, S), !.
bubbleSort(L, L).
```

5.2. Ներդրումների մեթոդ

Ներդրումներով տեսակավորման ալգորիթմը սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ ցուցակը տեսակավորված է,
- ոչ դատարկ ցուցակի տեսակավորման արդյունքն է այն ցուցակը, որը ստացվում է ցուցակի պոչը ներդրումների մեթոդով տեսակավորելու, այնուհետև ցուցակի գլուխը ստացված տեսակավորված ցուցակի մեջ ներդնելու արդյունքում:

Նշանակենք **insert(X, L1, L)** պնդումը, ըստ որի՝ **L** ցուցակը ստացվում է **L1** տեսակավորված ցուցակի համապատասխան դիրքում **X** տարրը ներդնելու արդյունքում: **insert** պրեդիկատը սահմանենք հետևյալ կերպ.

```
insert(X, [], [X]):- !.
insert(X, [X1|L1], [X, X1|L1]):- X=<X1, !.
insert(X, [X1|L1], [X1|L]):- insert(X, L1, L).
```

Հիմք ընդունելով **insert** պրեդիկատը՝ ներդրումներով տեսակավորման ալգորիթմը ներկայացնենք հետևյալ կերպ.

```
insertionSort([], []):- !.
insertionSort([X|L], S):- insertionSort(L, S1), insert(X, S1, S).
```

5.3. Արագ տեսակավորում

Արագ տեսակավորման ալգորիթմը սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ ցուցակը տեսակավորված է,
- ոչ դատարկ ցուցակի տեսակավորման արդյունքն է այն ցուցակը, որը ստացվում է ցուցակի գլխից փոքր և մեծ կամ հավասար տարրերից բաղկացած ցուցակները արագ տեսակա-

վորման ալգորիթմով տեսակավորելու և ցուցակի գլուխը դրանց միջև գրառելու արդյունքում:

Նշանակենք **partition(L, P, L1, L2)** պնդումը, ըստ որի՝ **L1** և **L2** ցուցակները կազմված են **L** ցուցակի համապատասխանաբար **P**-ից փոքր և **P**-ից մեծ կամ հավասար տարրերից: **partition** պրեդիկատը սահմանենք հետևյալ կերպ.

```
partition([], _, [], []):- !.
partition([X|L], P, [X|L1], L2):- X<P, partition(L, P, L1, L2),!.
partition([X|L], P, L1, [X|L2]):- partition(L, P, L1, L2).
```

Հիմք ընդունելով **partition** պրեդիկատը՝ արագ տեսակավորման ալգորիթմը ներկայացնենք հետևյալ կերպ.

```
quickSort([], []):-!.
quickSort([X|L], S):-partition(L, X, L1, L2),
                              quickSort(L1, S1), quickSort(L2, S2),
append(S1, [X|S2], S).
```

5.4. Զուլումներով տեսակավորում

Զուլումներով տեսակավորման ալգորիթմը սահմանենք հետևյալ անդրադարձ եղանակով.

- դատարկ կամ մեկ տարր պարունակող ցուցակը տեսակավորված է,
- առնվազն երկու տարր պարունակող ցուցակի տեսակավորման արդյունքն է այն ցուցակը, որը ստացվում է տրված ցուցակը մեկ տարրի ճշտությամբ հավասար մասերի բաժանելու, այդ ցուցակները ձուլումների մեթոդով տեսակավորելու և ստացված տեսակավորված ցուցակները միաձուլելու արդյունքում:

Նշանակենք **split(L, L1, L2)** պնդումը, ըստ որի՝ **L1** և **L2** ցուցակները կազմված են **L** ցուցակի համապատասխանաբար կենտ և զույգ դիր-

քերում գրառված տարրերից: **split** պրեդիկատը սահմանենք հետևյալ կերպ.

```
split([], [], []):-!.
split([X], [X], []):-!.
split([X, Y|L], [X|L1], [Y|L2]):- split(L, L1, L2).
```

Հիմք ընդունելով **split** պրեդիկատը՝ ձուլումներով տեսակավորման ալգորիթմը ներկայացնենք հետևյալ կերպ.

```
mergeSort([], []):-!.
mergeSort([X], [X]):-!.
mergeSort(L, S):- split(L, L1, L2),
                  mergeSort(L1, S1), mergeSort(L2, S2),
                  merge(S1, S2, S).
```

Խնդիրներ

1. Ամբողջ թվերի հաջորդականությունը տեսակավորել հետևյալ կերպ. պահպանելով թվերի հարաբերական կարգը՝ նախ գրառել հաջորդականության բացասական, այնուհետև՝ զրոյի հավասար և վերջապես դրական անդամները: Օրինակ՝

   -5, 3, 0, -1, 0, 2, -2

   Հաջորդականությունը նման եղանակով տեսակավորելու արդյունքում կստանանք

   -5, -1, -2, 0, 0, 3, 2

   հաջորդականությունը:

2. Գտնել բնական թվերից կառուցված հաջորդականության որևէ ամենաերկար աճող ենթահաջորդականություն: Օրինակ՝

   6, 4, 5, 2, 8, 3, 12, 5, 9, 10

հաջորդականության ամենաերկար աճող ենթահաջորդականություններն են

4, 5, 8, 9, 10 և

2, 3, 5, 9, 10

հաջորդականությունները:
Առաջարկել այս խնդրի այնպիսի լուծում, որում յուրաքանչյուր ենթախնդիր լուծվում է մեկ անգամից ոչ ավելի:

6. Գործողություններ նոսր բազմանդամների հետ

x փոփոխականից կախված բազմանդամն անվանենք *նոսր*, եթե դրա ոչ զրոյական միանդամների թիվը փոքր է բազմանդամի աստիճանի համեմատ: Մենք կենթադրենք, որ այս բաժնում դիտարկվող բազմանդամները նոսր են՝ չնշելով «նոսր» բառը:

Ընտրենք բազմանդամների հետևյալ ներկայացումը:

Ենթադրենք, որ **C** իրական գործակցով և **K** աստիճանի ցուցչով **x** փոփոխականից կախված միանդամը ներկայացված է **x(C, K)** կառուցվածքի միջոցով: Բազմանդամը ներկայացնենք աստիճանների նվազման կարգով ընթացող իր ոչ զրոյական միանդամների ցուցակով: Օրինակ՝ $\mathbf{5.5x^{100} - 3.5x^{50} + 10.0}$ բազմանդամի ներկայացումն է.

[x(5.5, 100), x(-3.5, 50), x(10.0, 0)]:

Նշենք, որ **0** բազմանդամը կներկայացվի դատարկ ցուցակով:

Կառուցենք *նոսր բազմանդամ* տվյալների վերացական տիպ, որի արժեքներն են ցուցակների տեսքով ներկայացված բազմանդամները: Տիպի գործողությունների ցանկում ընդգրկենք բազմանդամների նկատմամբ կիրառվող այնպիսի գործողություններ, ինչպիսիք են ածանցումը, գումարումը, հանումը, բազմապատկումը, բաժանման քանորդի/մնացորդի որոշումը:

Ստորև դիտարկենք վերը նշված գործողությունների իրականացումը **Prolog** լեզվով:

6.1. Ածանցում

Բազմանդամի ածանցյալը սահմանենք հետևյալ անդրադարձ եղանակով.

- ոչ հաստատուն միանդամի ածանցյալն է այն միանդամը, որի գործակիցն է տրված միանդամի գործակցի և աստիճանի

ցուցչի արտադրյալը, իսկ աստիճանի ցուցիչն է տրված միանդամի աստիճանի ցուցիչը՝ մեկով պակաս վերցված,

- դատարկ կամ հաստատուն միանդամից բաղկացած բազմանդամի ածանցյալն է դատարկ ցուցակ-բազմանդամը:
- ոչ հաստատուն ավագ միանդամ ունեցող բազմանդամի ածանցյալն է այն ցուցակ-բազմանդամը, որի գլուխն է տրված բազմանդամի գլխի ածանցյալը, պոչն է տրված բազմանդամի պոչի ածանցյալը:

Նշանակենք **derivation(P, P1)** պնդումը, ըստ որի՝ **P** բազմանդամի ածանցյալն է **P1**-ը: **derivation** պրեդիկատը սահմանենք հետևյալ կերպ.

```
derivation([],[]):- !.
derivation([x(_, 0)],[]):- !.
derivation([x(C, K)|P],[x(C1, K1)|P1]):-
                        C1 is C*K, K1 is K-1, derivation(P, P1).
```

6.2. Գումարում

Քանի որ բազմանդամները ոչ զրոյական միանդամների կարգավորված ցուցակներ են, ուստի բազմանդամների գումարման խնդիրը վերածվում է կարգավորված ցուցակների միաձուլման խնդրին այն տարբերությամբ, որ հավասար աստիճան ունեցող միանդամներ հանդիպելիս պատասխան ցուցակում անհրաժեշտ է ընդգրկել դրանց գումարը ներկայացնող միանդամը, եթե այն զրո չէ (այն է՝ միանդամների գործակիցները գումարվում են, ցուցիչը մնում է նույնը):

Նշանակենք **addition(P1, P2, P)** պնդումը, ըստ որի՝ **P1** և **P2** բազմանդամների գումարն է **P**-ն: **addittion** պրեդիկատը սահմանենք հետևյալ կերպ.

```
addition([], P, P):- !.
addition(P, [], P):- !.
addition([x(C1, K1)|P1], [x(C2, K2)|P2], [x(C1, K1)|P]):-
                                K1>K2, addition(P1, [x(C2, K2)|P2], P), !.
addition([x(C1, K1)|P1], [x(C2, K2)|P2], [x(C2, K2)|P]):-
                                K2>K1, addition([x(C1, K1)|P1], P2, P), !.
addition([x(C1, K)|P1], [x(C2, K)|P2], [x(C, K)|P]):-
                        C is C1+C2, abs(C)>0.000001, addition(P1, P2, P), !.
addition([_|P1], [_|P2], P):- addition(P1, P2, P).
```

Հաշվի առնելով իրական թվերի մոտավոր ներկայացումը համակարգչում՝ **C** թվի համեմատությունը **0**-ի հետ այս ծրագրում փոխարինված է **abs(C)>0.000001** համեմատությամբ:

6.3. Հանում

Բազմանդամների հանման խնդիրը բերվում է առաջին բազմանդամին բացասական նշանով երկրորդ բազմանդամի գումարման խնդրին: Նշանակենք **subtraction(P1, P2, P)** պնդումը, ըստ որի՝ **P1** և **P2** բազմանդամների տարբերությունն է **P**-ն: **subtraction** պրեդիկատը սահմանենք հետևյալ կերպ.

```
subtraction(P1, P2, P):- changeSign(P2, P3), addition(P1, P3, P).
```

Ըստ այստեղ օգտագործված **changeSign(P1, P2)** պնդման՝ **P2** բազմանդամը ստացվում է **P1** բազմանդամից նշանը շրջելու արդյունքում: **changeSign** պրեդիկատի սահմանումն է.

```
changeSign([], []):-!.
changeSign([x(C1, K)|P1], [x(C2, K)|P2]):-
                                C2 is -C1, changeSign(P1, P2).
```

6.4. Բազմապատկում

Դիտարկենք երկու բազմանդամների (դիցուք՝ **P1** և **P2**) բազմապատկման հետևյալ անդրադարձ ալգորիթմը.

- եթե **P1=0**, ապա բազմապատկման արդյունքն է **0**-ն,
- հակառակ դեպքում բազմապատկման արդյունքը սահմանվում է հետևյալ կերպ. եթե առաջին բազմանդամի ավագ միանդամն է **M**-ը, իսկ մնացած միանդամներից կազմված բազմանդամն է **Q**-ն, ապա բազմապատկման արդյունքն է **M*P2 + Q*P2** բազմանդամը:

Նշանակենք **multiplication(P1, P2, P)** պնդումը, ըստ որի՝ **P1** և **P2** բազմանդամների արտադրյալն է **P**-ն: **multiplication** պրեդիկատը սահմանենք հետևյալ կերպ.

```
multiplication([],_,[]):- !.
multiplication([x(C1, K1)|P1],P2, P):-
                        multiplication1(x(C1, K1), P2, Q1),
                        multiplication(P1, P2, Q2),
                        addition(Q1, Q2, P).
```

Ըստ **multiplication1(M, P1, P)** պնդման՝ **P** բազմանդամը **M** միանդամի և **P1** բազմանդամի արտադրյալն է: **multiplication1** պրեդիկատի սահմանումն է.

```
multiplication1(_,[], []):-!.
multiplication1(x(C1, K1), [x(C2, K2)|P2], [x(C, K)|P]):-
        C is C1*C2, K is K1+K2, multiplication1(x(C1, K1), P2, P).
```

6.5. Բաժանման քանորդի որոշում

Դիտարկենք երկու բազմանդամների (դիցուք՝ **P1** և **P2**) բաժանման քանորդի որոշման հետևյալ անդրադարձ ալգորիթմը.

- եթե **P1=0** կամ **P1**-ի աստիճանը փոքր է **P2**-ի աստիճանից, ապա բաժանման քանորդն է **0**-ն,

- հակառակ դեպքում բաժանման քանորդը սահմանվում է հետևյալ կերպ. եթե բազմանդամների ավագ միանդամներն են **M1**-ն ու **M2**-ը, ապա բաժանման քանորդի ավագ միանդամն է **M=M1/M2**-ը, քանորդի մնացած միանդամների գումարն է **P1-M*P2** բազմանդամի բաժանման քանորդը **P2**-ի վրա:

Նշանակենք **division(P1, P2, P)** պնդումը, ըստ որի՝ **P2** բազմանդամի վրա **P1** բազմանդամի բաժանման քանորդն է **P**-ն: **division** պրեդիկատի սահմանումն է.

```
division([],_, []):- !.
division([x(_, K1)|_], [x(_, K2)|_], []):-  K1<K2, !.
division([x(C1, K1)|P1], [x(C2, K2)|P2], [x(C, K)|P]):-
                    C is C1/C2, K is K1-K2,
                    multiplication1(x(C, K), [x(C2, K2)|P2], Q),
                    subtraction([x(C1, K1)|P1], Q, R),
                    division(R, [x(C2, K2)|P2], P).
```

6.6. Բաժանման մնացորդի որոշում

Երկու բազմանդամների (դիցուք՝ **P1** և **P2**) բաժանման մնացորդը ներկայացնենք հետևյալ բանաձևի միջոցով.

P1-P2*(P1//P2),

որտեղ **(P1//P2)**-ով նշանակված է **P2** բազմանդամի վրա **P1** բազմանդամի բաժանման քանորդը:

Նշանակենք **modulus(P1, P2, P)** պնդումը, ըստ որի՝ **P2** բազմանդամի վրա **P1** բազմանդամի բաժանման մնացորդն է **P**-ն: Վերը նշված բանաձևից բխում է **modulus** պրեդիկատի հետևյալ սահմանումը.

```
modulus(P1, P2, P):- division(P1, P2, Q),
                     multiplication(P2, Q, R),
                     subtraction(P1, R, P).
```

Խնդիրներ

1. Հաշվել բազմանդամի արժեքը տրված կետում:
2. Տրված են **[a, b]** միջակայքը, **ε>0** իրական թիվը և **P** բազմանդամը, որը փոխում է նշանը **[a, b]** միջակայքի եզրերում: Կիսման մեթոդի օգտագործմամբ **ε** ճշտությամբ որոշել **P** բազմանդամի արմատը **[a, b]** միջակայքում:

7. Տողերի մշակում

Տողը սահմանենք որպես սիմվոլների հաջորդականություն: Տողի բացատանիշից տարբերվող, իրար հաջորդող սիմվոլների ցանկացած ենթահաջորդականություն անվանենք *բառ*, եթե այն սահմանափակված է տողի սկզբով, տողի վերջով կամ բացատանիշ(ներ)ով: Օրինակ, եթե բացատանիշը նշանակենք **Ա**-ով, ապա

Աa(1)ԱԱbetaԱ;

տողի բառերն են **a(1)**-ը, **beta**-ն և **;**-ը:

Տողը ներկայացնենք սիմվոլներին համապատասխանող ատոմների ցուցակի տեսքով՝ սահմանելով սիմվոլների և ատոմների համապատասխանությունը հետևյալ կերպ.

- լատինական փոքրատառին կամ տասական թվանշանին համապատասխանող ատոմն է նույն այդ սիմվոլը,
- լատինական փոքրատառից և տասական թվանշանից տարբերվող սիմվոլին համապատասխանող ատոմն է ապաթարցերի մեջ վերցված սիմվոլը:

Օրինակ՝ վերը նշված տողին համապատասխանում է հետևյալ ցուցակը.

['Ա',a,'(',1,')','Ա','Ա',b,e,t,a,'Ա',';']:

Ստորև դիտարկենք տողերի մշակման մի քանի խնդիրներ:

7.1. Միատեսակ փակագծերի հաշվեկշռի ստուգում

Կասենք, որ տողում առկա է (կլոր) փակագծերի հաշվեկշիռ, եթե.

- տողում բացող փակագծերի քանակը հավասար է փակող փակագծերի քանակին, և

- տողի ցանկացած վերջնամասում փակող փակագծերի քանակը մեծ է կամ հավասար բացող փակագծերի քանակին:

Փակագծերի հաշվեկշռի ստուգման խնդիրը լուծելու համար տողի յուրաքանչյուր վերջածանցին (վերջնամասին) համապատասխանեցնենք դրա փակող և բացող փակագծերի քանակների *տարբերությունը*: Նկատենք, որ փակագծերի հաշվեկշիռ ունեցող տողի համար այս արժեքը զրո է, իսկ դրա յուրաքանչյուր վերջածանցի համար՝ ոչ բացասական:

Տող և *ոչ բացասական թիվ* զույգի համար սահմանենք փակագծերի հաշվեկշռի առկայություն հետևյալ կերպ.

- *դատարկ տող* և *զրո* զույգի համար առկա է փակագծերի հաշվեկշիռ,
- *բացող փակագծով սկսվող տող* և *ոչ բացասական թիվ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե այն առկա է *տողի պոչ* և *մեկով մեծ թիվ* զույգի համար,
- *փակող փակագծով սկսվող տող* և *դրական թիվ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե այն առկա է *տողի պոչ* և *մեկով փոքր թիվ* զույգի համար,
- *բացող և փակող փակագծերից տարբերվող սիմվոլով սկսվող տող* և *ոչ բացասական թիվ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե այն առկա է *տողի պոչ* և *սկզբնական թիվ* զույգի համար:

Դիցուք **checkParentheses(S)**-ը պնդում է, ըստ որի՝ **S** տողում առկա է փակագծերի հաշվեկշիռ: **checkParentheses** պրեդիկատը ներկայացնենք հետևյալ կերպ.

checkParentheses(S):- checkParentheses1(S, 0).

Այստեղ **checkParentheses1(S, D)**-ն պնդում է, ըստ որի՝ **S** տող և **D** ոչ բացասական թիվ զույգի համար առկա է փակագծերի հաշվեկշիռ: Հիմք ընդունելքվ վերը բերված ալգորիթմը՝ **checkParentheses1** պրեդիկատը սահմանենք հետևյալ կերպ.

```
checkParentheses1([], 0):-          !.
checkParentheses1(['('|S], D):-     !,
                    checkParentheses1(S, D1), D1>0, D is D1-1.
checkParentheses1([')'|S], D):-     !,
                    checkParentheses1(S, D1), D is D1+1.
checkParentheses1([_|S], D):-
                    checkParentheses1(S, D).
```

7.2. Եռատեսակ փակագծերի հաշվեկշռի ստուգում

Գասենք, որ տողում առկա է եռատեսակ (դիցուք՝ *կլոր*, *քառակուսի* և *ձնավոր*) փակագծերի հաշվեկշիռ, եթե.

- տողում չկան նշված տեսակի փակագծեր,
- տողը ստացվում է՝ եռատեսակ փակագծերի հաշվեկշիռ ունեցող տողից վերցնելով այն այս կամ այն տեսակի փակագծերի մեջ,
- տողը ստացվում է եռատեսակ փակագծերի հաշվեկշիռ ունեցող երկու տողերից դրանք միմյանց կցագրելու արդյունքում:

Եռատեսակ փակագծերի հաշվեկշռի ստուգման խնդիրը լուծելու համար ներմուծենք պահունակ, որտեղ կհիշենք տողը մշակելու ընթացքում հանդիպած բացող փակագծերը: *Տող* և *պահունակ* զույգի համար սահմանենք փակագծերի հաշվեկշռի առկայություն հետևյալ կերպ.

- *դատարկ տող* և *դատարկ պահունակ* զույգի համար առկա է փակագծերի հաշվեկշիռ,
- *բացող փակագծով սկսվող տող* և *պահունակ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե այն առկա է *տողի պոչ* և *ընդլայնված պահունակ* զույգի համար, որտեղ ընդլայնված պահունակը ստացվում է տողի առջևում գրառված բացող փակագիծը պահունակին ավելացնելու արդյունքում,

- *փակող փակագծով սկսվող տող* և *ոչ դատարկ պահունակ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե.
  - պահունակի գագաթային սիմվոլն է տողի առջևում գրառված փակող փակագծին համապատասխանող բացող փակագիծը,
  - առկա է փակագծերի հաշվեկշիռ *տողի պոչ* և *կրճատված պահունակ* զույգի համար, որտեղ կրճատված պահունակը ստացվում է գագաթային սիմվոլը պահունակից հեռացնելու արդյունքում,
- *փակագծից տարբերվող սիմվոլով սկսվող տող* և *պահունակ* զույգի համար առկա է փակագծերի հաշվեկշիռ, եթե այն առկա է *տողի պոչ* և *սկզբնական պահունակ* զույգի համար:

Նշանակենք **checkBrackets(S)** պնդումը, ըստ որի՝ **S** տողում առկա է եռատեսակ փակագծերի հաշվեկշիռ: **checkBrackets** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
checkBrackets(S):-checkBrackets1(S, []).
```

Ըստ **checkBrackets1(S, Stack)** պնդման **S** տող և **Stack** պահունակ զույգի համար առկա է փակագծերի հաշվեկշիռ: Հիմք ընդունելով փակագծերի հաշվեկշռի ստուգման վերը բերված ալգորիթմը՝ **checkBrackets1** պրեդիկատը սահմանենք հետևյալ կերպ.

```
checkBrackets1([], []):- !.

checkBrackets1(['('|S], Stack):-
                        checkBrackets1(S, ['('|Stack]), !.
checkBrackets1(['['|S], Stack):-
                        checkBrackets1(S, ['['|Stack]), !.
checkBrackets1(['{'|S], Stack):-
                        checkBrackets1(S, ['{'|Stack]), !.

checkBrackets1([')'|S], ['('|Stack]):-
                        checkBrackets1(S,Stack), !.
```

```
checkBrackets1([']'|S], ['['|Stack]):-
                        checkBrackets1(S,Stack), !.
checkBrackets1(['}'|S], ['{'|Stack]):-
                        checkBrackets1(S,Stack), !.

checkBrackets1([')'|_], _):-        !, fail.
checkBrackets1([']'|_], _):-        !, fail.
checkBrackets1(['}'|_], _):-        !, fail.

checkBrackets1([_|S], Stack):-  checkBrackets1(S,Stack).
```

7.3. Բառերի քանակի որոշում

Բառերի քանակը տողում սահմանենք հետևյալ կերպ.

- դատարկ տողում բառերի քանակն է **0**-ն,
- բացատանիշով սկսվող տողում բառերի քանակն է բառերի քանակը տողի պոչում,
- բացատանիշից տարբերվող սիմվոլով (այն է՝ բառով) սկսվող տողում բառերի քանակը մեկով ավելի է բառերի քանակից այն տողում, որը ստացվում է տրված տողից առաջին բառը հեռացնելու արդյունքում:

Նշանակենք **numberOfWords(S, N)** պնդումը, ըստ որի՝ **S** տողի բառերի քանակն է **N**-ը: Հիմք ընդունելով տողերի քանակի վերը բերված սահմանումը՝ **numberOfWords** պրեդիկատը նկարագրենք հետևյալ կերպ.

```
numberOfWords([], 0):-  !.
numberOfWords([' '|S], K):-  numberOfWords(S, K), !.
numberOfWords(S, K):-        skipFirstWord(S, S1),
                             numberOfWords(S1, K1), K is K1+1.
```

Այստեղ, ըստ **skipFirstWord(S, S1)** պնդման, **S1** տողը ստացվում է բառով սկսվող **S** տողի առաջին բառը հեռացնելու արդյունքում: **skipFirstWord** պրեդիկատը սահմանենք հետևյալ կերպ.

```
skipFirstWord([X|S], S1):-X\=' ', skipFirstWord(S, S1), !.
skipFirstWord(S, S).
```

Խնդիրներ

1. Տրված են **S** տողը և **N** բնական թիվը: Դարձնել **S** տողի երկարությունը հավասար **N**-ի հետևյալ կերպ. եթե **S** տողի երկարությունը փոքր է **N**-ից, ապա տողի վերջում ավելացնել համապատասխան քանակությամբ բացատանիշեր: Հակառակ դեպքում կրճատել տողը՝ թողնելով դրա առաջին **N** սիմվոլները:
2. Ավելացնել տրված տողի վերջում մինիմալ թվով սիմվոլներ այնպես, որ արդյունքում ստացվի պալինդրոմ:
3. Տրված են **S** տողը և **K** բնական թիվը: Կառուցել նոր տող՝ ընդգրկելով դրանում **S** տողի **K** երկարություն ունեցող բառերը՝ անջատելով դրանք միմյանցից բացատանիշով: **S** տողում **K** երկարություն ունեցող բառերի բացակայության դեպքում ստանալ դատարկ տող:

8. Գործողություններ մատրիցների հետ

A={a_{ij}}, **1≤i≤m, 1≤j≤n, m×n** չափի մատրիցը սահմանվում է որպես տարրերի աղյուսակ, որի տողերն ունեն **n**, իսկ սյունակները՝ **m** երկարություն.

$$\mathbf{A} = \begin{pmatrix} a_{11} \cdots a_{1n} \\ \cdots \\ a_{m1} \cdots a_{mn} \end{pmatrix}$$

Պայմանավորվենք մատրիցի տողերը ներկայացնել տարրերի ցուցակների, իսկ մատրիցը՝ տողերի ցուցակի տեսքով.

[[a_{11}, … , a_{1n}], …, [a_{m1}, … , a_{mn}]]

Օրինակ՝

$$\begin{pmatrix} 2 & 3 & 4 \\ -1 & 0 & 5 \end{pmatrix}$$

2×3 չափի մատրիցի ներկայացումն է

[[2, 3, 4], [-1, 0, 5]]

ցուցակների ցուցակը:

Հիմք ընդունելով այս ներկայացումը՝ կառուցենք *մատրից* տվյալների վերացական տիպ՝ ներառելով դրանում մատրիցների մշակման այնպիսի գործողություններ, ինչպիսիք են անկյունագծի որոշումը, տրանսպոնացումը, դետերմինանտի հաշվումը:

Ստորև դիտարկենք այս գործողությունների իրականացումը **Prolog** լեզվով:

8.1. Անկյունագծի որոշում

n×n չափի (քառակուսի) մատրիցի գլխավոր անկյունագիծը սահմանենք որպես

$$[a_{11}, \dots , a_{nn}]$$

ցուցակ:

Դիտարկենք քառակուսի մատրիցի անկյունագծի որոշման հետևյալ անդրադարձ ալգորիթմը.

- դատարկ մատրիցի անկյունագիծն է դատարկ ցուցակը,
- ոչ դատարկ մատրիցի անկյունագիծն է այն ցուցակը, որի գլուխն է մատրիցի առաջին տողի առաջին տարրը, պոչն է տրված մատրիցի առաջին տողը և առաջին սյունակը հեռացնելու արդյունքում ստացված քառակուսի մատրիցի անկյունագիծը:

Նշանակենք **diagonal(M, D)** պնդումը, ըստ որի՝ **D**-ն **M** քառակուսի մատրիցի անկյունագիծն է: **diagonal** պրեդիկատը սահմանենք հետևյալ կերպ.

```
diagonal([], []):-                !.
diagonal([[X|_]|M], [X|D]):- removeFirstColumn(M, M1),
diagonal(M1, D).
```

Այստեղ **removeFirstColumn(M, M1)**-ը պնդումն է, ըստ որի՝ **M1** մատրիցը ստացվում է **M** մատրիցի առաջին սյունակը հեռացնելու արդյունքում: Այս գործողությունը նկարագրենք հետևյալ անդրադարձ եղանակով.

- դատարկ կամ մեկ սյունակ ունեցող մատրիցի առաջին սյունակը հեռացնելու արդյունքում ստացվում է դատարկ մատրից,
- առնվազն երկու սյունակ ունեցող մատրիցի առաջին սյունակը հեռացնելու արդյունքում ստացվում է մատրից (տողերի ցուցակ), որի գլուխն է տրված մատրիցի առաջին տողի պոչը,

պոչն է տրված մատրիցից առաջին տողը և առաջին սյունակը հեռացնելու արդյունքում ստացված մատրիցը:

removeFirstColumn պրեդիկատը սահմանենք հետևյալ կերպ.

```
removeFirstColumn([], []):-          !.
removeFirstColumn([[_]|_], []):-     !.
removeFirstColumn([[_|R]|M], [R|M1]):-
                                     removeFirstColumn(M, M1).
```

8.2. Տրանսպոնացում

A={a_{ij}}, 1≤i≤m, 1≤j≤n, m×n չափի մատրիցի տրանսպոնացումը սահմանվում է որպես **A'={a_{ji}}, 1≤j≤n, 1≤i≤m, n×m** չափի մատրից:

Դիտարկենք տրանսպոնացված մատրիցի կառուցման հետևյալ անդրադարձ ալգորիթմը.

- դատարկ մատրիցի տրանսպոնացումն է դատարկ մատրիցը,
- ոչ դատարկ մատրիցի տրանսպոնացումն է տրանսպոնացված առաջին տողի կցագրումը մատրիցի առաջին տողը հեռացնելուց հետո ստացված մատրիցի տրանսպոնացմանը:

Նշանակենք **transposition(M, M1)** պնդումը, ըստ որի՝ **M1** մատրիցը **M** մատրիցի տրանսպոնացումն է: **transposition** պրեդիկատը սահմանենք հետևյալ կերպ.

```
transposition([], []):-          !.
transposition([R|M], M1):- createColumn(R, C),
                           transposition(M,M2),
                           addColumnToMatrix(C, M2, M1).
```

Այստեղ ըստ **createColumn(R, C)** պնդման, **C** սյունակը ստացվում է **R** տողի տրանսպոնացման արդյունքում, իսկ ըստ **addMatrixToColumn(C, M1, M)**-ը՝ պնդման **M** մատրիցը ստացվում է **C** սյունակը **M1** մատրիցին կցագրելու արդյունքում: Ստորև բերված են **createColumn** և **addMatrixToColumn** պրեդիկատների սահմանումները.

```
createColumn([], []):-          !.
createColumn([X|R], [[X]|C]):-  createColumn(R, C).

addColumnToMatrix(C, [], C):-   !.
addColumnToMatrix([[X]|C], [R|M1], [[X|R]|M]):-
                        addColumnToMatrix(C, M1, M).
```

8.3. Դետերմինանտի հաշվում

n×n չափի **M** մատրիցի դետերմինանտը կարելի է հաշվել՝ վերլուծելով մատրիցն ըստ առաջին սյունակի.

$$\det(M) = \sum_{i=1}^{n} (-1)^{i+1} a_{i1} \cdot \det(M_{i1})$$

որտեղ M_{ij} մատրիցը ստացվում է **M** մատրիցից **i**-րդ տողը և **j**-րդ սյունակը հեռացնելու արդյունքում: Դետերմինանտի այս ներկայացումը հնարավորություն է տալիս **n×n** չափի մատրիցի դետերմինանտի հաշվումը բերելու **n** հատ **(n-1)×(n-1)** չափի մատրիցների դետերմինանտների հաշվման: Սակայն անմիջականորեն այս մոտեցումն իրականացնող ծրագիրը կլինի ոչ արդյունավետ միևնույն ենթամատրիցների դետերմինանտները բազմիցս հաշվելու պատճառով: Օրինակ՝ $(M_{11})_{11}=(M_{21})_{11}$ նույնությունից հետևում է, որ վերը նշված եղանակով մատրիցի դետերմինանտը հաշվելիս կրկնակի է հաշվվում տրված մատրիցի առաջին երկու տողերն ու սյունակները հեռացնելու արդյունքում ստացված ենթամատրիցի դետերմինանտը:

Ավելի արդյունավետ ծրագիր կառուցելու համար **n×n** չափի մատրիցի դետերմինանտի հաշվման խնդիրը բերենք ընդամենը մեկ **(n-1)×(n-1)** չափի մատրիցի դետերմինանտի հաշվման խնդրին: Դրա համար նկատենք, որ եթե մատրիցի առաջին սյունակի բոլոր տարրերը զրոներ են, ապա մատրիցի դետերմինանտը զրո է: Հակառակ դեպքում նախ վերադասավորենք մատրիցի տողերն այնպես, որ առաջին տողի առաջին տարրը դառնա ոչ զրոյական (նշենք, որ տո-

ղերի վերադասավորումը կարող է բերել դետերմինանտի նշանի փոփոխման), այնուհետև պահպանելով դետերմինանտի արժեքը՝ ձևափոխենք մատրիցն այնպես, որ առաջին սյունակի բոլոր տարրերը, բացի առաջինից, զրոյանան: Ստացված մատրիցի դետերմինանտը հաշվելու համար բավարար է բազմապատկել մատրիցի առաջին տողի առաջին տարրը դրա հանրահաշվական լրացման վրա:

Նշված ալգորիթմի կիրառումը հենվում է մատրիցների տեսությունից հայտնի հետևյալ փաստերի վրա.

- երկու տող (սյունակ) տեղափոխելու արդյունքում դետերմինանտի նշանը փոխվում է, սակայն բացարձակ արժեքը մնում է նույնը,
- թվով բազմապատկած տողը (սյունակը) մեկ այլ տողին (սյունակին) գումարելու դեպքում դետերմինանտի արժեքը մնում է նույնը:

Նշանակենք **determinant(M, D)** պնդումը, ըստ որի՝ **D**-ն **M** քառակուսի մատրիցի դետերմինանտն է: **determinant** պրեդիկատը սահմանենք հետևյալ կերպ.

```
determinant([[X]], X):-    !.
determinant(M, D):-        rearrangeRows(M, [[X|R]|M1], Sign),!,
                           reduceMatrix([X|R], M1, M2),
                           determinant(M2, D1), D is Sign*X*D1.
determinant(_, 0).
```

Այստեղ օգտագործված **rearrangeRows** և **reduceMatrix** պրեդիկատներն ունեն հետևյալ իմաստը.

- **rearrangeRows(M, M1, Sign)**-ը պնդում է, ըստ որի՝ **M** մատրիցում կան ոչ զրոյական տարրերով սկսվող տողեր, և **M1** մատրիցը ստացվում է **M** մատրիցից առաջին այդպիսի տողը հաջորդաբար կատարվող տեղափոխություններով վերևում տեղադրելու արդյունքում: Զույգ թվով տեղափոխությունների դեպքում **Sign=1**, այլապես **Sign=-1**,

- **reduceMatrix(R, M1, M2)** պրեդիկատի արգումենտներն են ոչ զրոյական տարրով սկսվող **n>0** երկարություն ունեցող **R** տողը և համապատասխանաբար **(n-1)×n** և **(n-1)×(n-1)** չափեր ունեցող **M1** և **M2** մատրիցները: Պրեդիկատը ներկայացնում է պնդում, ըստ որի՝ **M2** քառակուսի մատրիցը ստացվում է **R** տողի միջոցով **M1** մատրիցի առաջին սյունակը զրոյացնելու արդյունքում առանց այդ զրոյացված սյունակի:

rearrangeRows և **reduceMatrix** պրեդիկատները սահմանենք հետևյալ կերպ.

```
rearrangeRows([[Zero|_]], _, _):- abs(Zero)<0.0000001,!, fail.
rearrangeRows([[Zero|R]|M],[R1,[Zero|R]|M1], Sign):-
                        abs(Zero)<0.0000001,!,
                        rearrangeRows(M,[R1|M1], Sign1),
                        Sign is -Sign1.
rearrangeRows(M, M, 1).

reduceMatrix(_, [], []):- !.
reduceMatrix([X|R], [[X1|R1]|M], [R2|M1]):- C is -X1/X,
                                    combination(C, R, R1, R2),
                                    reduceMatrix([X|R], M, M1).
```

Այստեղ **combination(C, R, R1, R2)**-ը պնդում է, ըստ որի՝ **R2** տողը ստացվում է **R** տողը **C** հաստատունով բազմապատկելու և ստացված տողին **R1** տողը գումարելու արդյունքում, այն է՝ **R2=C·R+R1**: **combination** պրեդիկատի սահմանումն է.

```
combination(_, [], [], []):-!.
combination(C, [X|R], [X1|R1], [X2|R2]):-
                                    X2 is C*X+X1,
                                    combination(C, R, R1, R2).
```

Խնդիրներ

1. Տրված են **n×n** չափի **M** մատրիցը և **i, j** ամբողջ թվերը, որտեղ **1≤i,j≤n**: Կառուցել **M** մատրիցի a_{ij} տարրի հանրահաշվական լրացումը՝ $Aij=(-1)^{i+j} \cdot det(M_{ij})$, որտեղ M_{ij} մատրիցը ստացվում է **M** մատրիցից **i**–րդ տողը և **j**-րդ սյունակը հեռացնելու արդյունքում:
2. Տրված են **p×q** չափի **A** և **q×r** չափի **B** մատրիցները: Կառուցել **p×r** չափի **A×B** մատրիցը:
3. Տրված է **m×n** չափի **M** մատրիցը: Կառուցել **M** մատրիցը ժամացույցի սլաքի ուղղությամբ պարուրագծով շրջանցելու արդյունքում ստացվող ցուցակը: Ենթադրել, որ շրջանցումը սկսվում է առաջին տողի առաջին տարրից: Օրինակ՝

$$M = \begin{pmatrix} 2 & 3 & 4 \\ 1 & 0 & 3 \\ 7 & 3 & 2 \end{pmatrix}$$

մատրիցի նշված շրջանցման արդյունքն է

S=[2, 3, 4, 3, 2, 3, 7, 1, 0]

ցուցակը:

9. Բինար ծառեր

9.1. Բինար ծառի սահմանում

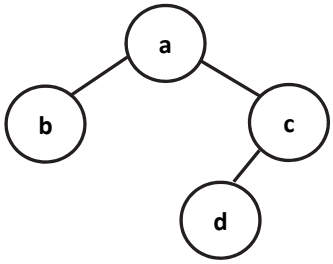
Բինար ծառը սահմանվում է որպես գագաթների այնպիսի բազմություն, որը

- դատարկ է կամ
- բաղկացած է *արմատ* կոչվող գագաթից և արմատի *ձախ* ու *աջ ենթածառեր* կոչվող գագաթների չհատվող ենթաբազմություններից: Արմատի ձախ ու աջ ենթածառերը կրկին բինար ծառեր են:

Հիմք ընդունելով այս սահմանումը՝ կառուցենք բինար ծառ տվյալների տիպ, որի արժեքների բազմությունը սահմանենք որպես **binaryTree** հետևյալ պրեդիկատին բավարարող թերմերի բազմություն.

```
binaryTree(nil):- !.
binaryTree(bTree(L, _, R)):- binaryTree(L), binaryTree(R).
```

Օրինակ՝ **Նկ. 9.1**-ում բերված բինար ծառին



Նկ. 9.1

համապատասխանում է

```
bTree(
        bTree(nil, b, nil),
        a,
        bTree(
                bTree(nil, d, nil),
                c,
                nil
        )
)
```

կառուցվածքը:

Դատարկ ձախ և աջ ենթածառեր ունեցող գագաթները կոչվում են *տերևներ*: **Նկ. 9.1**-ում պատկերված ծառի տերևներն են **b**-ն և **d**-ն:

9.2. Բինար ծառի շրջանցումներ

Բինար ծառի *ուղիղ* (**preorder**), *սիմետրիկ* (**inorder**) և *հակադարձ* (**postorder**) կարգով շրջանցումները սահմանվում են հետևյալ կերպ.

Ուղիղ կարգով շրջանցում: Եթե ծառը դատարկ չէ, ապա.

- «այցելել» արմատը,
- ուղիղ կարգով շրջանցել արմատի ձախ ենթածառը,
- ուղիղ կարգով շրջանցել արմատի աջ ենթածառը:

Օրինակ՝ **Նկ. 9.1**-ում բերված ծառը ուղիղ կարգով շրջանցելու արդյունքում ստացվում է գագաթների **a, b, c, d** հաջորդականությունը:

Սիմետրիկ կարգով շրջանցում: Եթե ծառը դատարկ չէ, ապա.

- սիմետրիկ կարգով շրջանցել արմատի ձախ ենթածառը,
- «այցելել» արմատը,
- սիմետրիկ կարգով շրջանցել արմատի աջ ենթածառը:

Օրինակ՝ **Նկ. 9.1**-ում բերված ծառը սիմետրիկ կարգով շրջանցելու արդյունքում ստացվում է գագաթների **b, a, d, c** հաջորդականությունը:

Հակադարձ կարգով շրջանցում: Եթե ծառը դատարկ չէ, ապա.

- հակադարձ կարգով շրջանցել արմատի ձախ ենթածառը,
- հակադարձ կարգով շրջանցել արմատի աջ ենթածառը,
- «այցելել» արմատը:

Օրինակ՝ **Նկ. 9.1**-ում բերված ծառը հակադարձ կարգով շրջանցելու արդյունքում ստացվում է գագաթների **b, d, c, a** հաջորդականությունը:

Դիցուք **preorder(T, S)**, **inorder(T, S)**, **postorder(T, S)** պնդումներ են, ըստ որոնց՝ **T** բինար ծառը համապատասխանաբար ուղիղ, սիմետրիկ և հակադարձ կարգով շրջանցելու արդյունքում ստացվում է **S** ցուցակը: Նշված պրեդիկատների սահմանումներն են.

```
preorder(nil, []):- !.
preorder(bTree(L, X, R), [X|S]):-
        preorder(L, S1), preorder(R, S2), append(S1, S2, S).

inorder(nil, []):- !.
inorder(bTree(L, X, R), S):-
        inorder(L, S1), inorder(R, S2), append(S1, [X|S2], S).

postorder(nil, []):- !.
postorder(bTree(L, X, R), S):-
                postorder(L, S1), postorder(R, S2),
                append(S1, S2, S3), append(S3, [X], S).
```

9.3. Բինար ծառերի մշակման խնդիրներ

Կառուցենք բինար ծառի մի քանի բնութագրիչների որոշման ծրագրեր **Prolog** լեզվով:

9.3.1. Ծավալի հաշվում

Բինար ծառի ծավալը (գագաթների քանակը) նկարագրվում է հետևյալ անդրադարձ եղանակով.

- դատարկ ծառի ծավալը **0** է,
- ոչ դատարկ ծառի ծավալը մեկով ավելի է արմատի ձախ և աջ ենթածառերի ծավալների գումարից:

Դիցուք **size(T, N)**-ը պնդում է, ըստ որի՝ **T** բինար ծառի ծավալը **N** է: Ծավալի որոշման անդրադարձ նկարագրին համապատասխանում է **size** պրեդիկատի հետևյալ սահմանումը.

```
size(nil, 0):- !.
size(bTree(L, _, R), N):- size(L, N1), size(R, N2), N is N1+N2+1.
```

9.3.2. Բարձրության հաշվում

Բինար ծառի բարձրությունը սահմանվում է որպես արմատից դեպի տերև տանող ամենաերկար պարզ ճանապարհի երկարություն: Բինար ծառի բարձրությունը կարող ենք որոշել հետևյալ անդրադարձ եղանակով.

- դատարկ ծառի բարձրությունը **-1** է,
- ոչ դատարկ ծառի բարձրությունը մեկով ավելի է ձախ և աջ ենթածառերի բարձրություններից առավելագույնից:

Դիցուք **height(T, N)**-ը պնդում է, ըստ որի՝ **T** բինար ծառի բարձրությունը **N** է: Բարձրության որոշման անդրադարձ նկարագրին համապատասխանում է **size** պրեդիկատի հետևյալ սահմանումը.

```
height(nil, -1):- !.
height(bTree(L, _, R), N):- height(L, N1), height(R, N2),
                            max(N1, N2, M), N is M+1.
```

Այստեղ **max(N1, N2, N)**-ը պնդում է, ըստ որի՝ **N**-ը **N1** և **N2** թվերից առավելագույնն է: **max** պրեդիկատի սահմանումն է.

```
max(N1, N2, N1):- N1>=N2, !.
max(_, N2, N2).
```

9.3.3. Տերևների քանակի հաշվում

Բինար ծառի տերևների քանակը կարող ենք հաշվել հետևյալ անդրադարձ եղանակով.

- դատարկ ծառի տերևների քանակը **0** է,
- մեկ գագաթից բաղկացած ծառի տերևների քանակը **1** է,
- մեկից ավելի գագաթ ունեցող ծառի տերևների քանակն է ձախ և աջ ենթածառերի տերևների քանակների գումարը:

Նշանակենք **numberOfLeaves(T, N)** պնդումը, ըստ որի՝ որ **T** բինար ծառի տերևների քանակը **N** է: Տերևների քանակի որոշման անդրադարձ նկարագրին համապատասխանում է **numberOfLeaves** պրեդիկատի հետևյալ սահմանումը.

```
numberOfLeaves(nil, 0):- ! .
numberOfLeaves(bTree(nil, _, nil), 1):- !.
numberOfLeaves(bTree(L, _, R), N):-
    numberOfLeaves(L, N1),  numberOfLeaves(R, N2), N is N1+N2.
```

Խնդիրներ

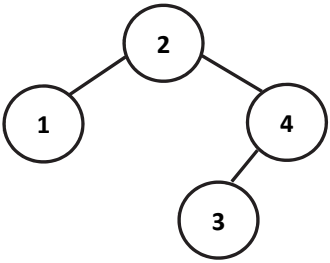
1. Վերականգնել բինար ծառը՝ հենվելով դրա
   ա) ուղիղ և սիմետրիկ կարգով շրջանցումների արդյունքում ստացված հաջորդականությունների վրա,
   բ) հակադարձ և սիմետրիկ կարգով շրջանցումների արդյունքում ստացված հաջորդականությունների վրա:
2. Որոշել բինար ծառի արմատից տերև տանող ամենաերկար ճանապարհներից որևէ մեկը:
3. Որոշել բինար ծառի լայնությունը, այն է՝ միևնույն մակարդակի վրա գտնվող գագաթների մաքսիմալ քանակը:

10. Որոնման բինար ծառեր

10.1. Որոնման բինար ծառի սահմանում

Բինար ծառը կոչվում է *որոնման բինար ծառ* (կամ պարզապես *որոնման ծառ*), եթե դրա գագաթները արժևորված են լրիվ կարգավորված բազմության տարրերով այնպես, որ յուրաքանչյուր գագաթի արժեքը մեծ է այդ գագաթի ձախ ենթածառի գագաթների արժեքներից և փոքր է կամ հավասար աջ ենթածառի գագաթների արժեքներից:

Օրինակ՝ որոնման ծառ է **Նկ. 10.1**-ում բերված ծառը.



Նկ. 10.1

10.2. Գործողություններ որոնման ծառերի հետ

Որոնման ծառերի նկատմամբ սահմանված հիմնական գործողություններն են տարրի *որոնման*, *ավելացման* և *հեռացման* գործողությունները, որոնց բարդությունը միջինում լոգարիթմորեն է կախված տարրերի քանակից: Դիտարկենք այս գործողությունների իրականացումը **Prolog** լեզվով:

10.2.1. Որոնում

Տարրի պատկանելիությունը որոնման ծառին սահմանենք հետևյալ անդրադարձ եղանակով. տարրը պատկանում է որոնման ծառին այն և միայն այն դեպքում, երբ այն.

- ծառի արմատի արժեքն է, կամ
- փոքր է ծառի արմատի արժեքից և պատկանում է արմատի ձախ ենթածառին, կամ
- մեծ է ծառի արմատի արժեքից և պատկանում է արմատի աջ ենթածառին:

Դիցուք **search(X, T)**-ն պնդում է, ըստ որի՝ **X** տարրը պատկանում է **T** որոնման ծառին: Պատկանելիության գործողության անդրադարձ նկարագրին համապատասխանում է **search** պրեդիկատի հետևյալ սահմանումը.

```
search(X, bTree(_, X, _)):- !.
search(X, bTree(L, Y, _) ):- X<Y, search(X, L), !.
search(X, bTree(_, _, R) ):- search(X, R).
```

10.2.2. Ավելացում

Տարրի (կրկնումներով) ավելացումը որոնման ծառին սահմանենք հետևյալ անդրադարձ եղանակով.

- տարրը դատարկ ծառին ավելացնելու դեպքում ստացվում է այդ տարրով արժևորված մեկ գագաթ ունեցող ծառ,
- տարրը ոչ դատարկ ծառին ավելացնելիս այն ավելացվում է արմատի ձախ ենթածառին, եթե այն փոքր է արմատի արժեքից և աջ ենթածառին՝ հակառակ դեպքում:

Դիցուք **insert(X, T, T1)**-ը պնդում է, ըստ որի՝ **T1** որոնման ծառը ստացվում է **X** տարրը **T** որոնման ծառին ավելացնելու արդյունքում: Ավելացման գործողության անդրադարձ նկարագրին համապատասխանում է **insert** պրեդիկատի հետևյալ սահմանումը.

```
insert(X, nil, bTree(nil, X, nil)):- !.
insert(X, bTree(L, Y, R), bTree(L1, Y, R)):- X<Y, insert(X, L, L1), !.
insert(X, bTree(L, Y, R), bTree(L, Y, R1)):- insert(X, R, R1).
```

10.2.3. Հեռացում

Տարրի առաջին հանդիպած արժեքի հեռացումը որոնման ծառից սահմանենք հետևյալ անդրադարձ եղանակով.

- տարրը դատարկ ծառից հեռացնելու դեպքում ստացվում է դատարկ ծառ,
- ոչ դատարկ ծառից արմատի արժեքից տարբերվող տարրի հեռացումը կատարվում է արմատի ձախ ենթածառից, եթե հեռացվող տարրը փոքր է արմատի արժեքից և արմատի աջ ենթածառից՝ հակառակ դեպքում:
- ոչ դատարկ ծառից արմատի արժեքի հեռացումը կարող է կատարվել հետևյալ կերպ.
  - եթե արմատի ենթածառերից որևէ մեկը դատարկ է, ապա արմատի արժեքը հեռացնելիս ստացվում է մյուս (հնարավոր է դատարկ) ենթածառը,
  - եթե արմատի ենթածառերը դատարկ չեն, ապա արմատի արժեքը հեռացնելիս արմատին վերագրվում է աջ ենթածառի փոքրագույն արժեքը, ինչից հետո այն հեռացվում է աջ ենթածառից (նշենք, որ արմատի աջ ենթածառում փոքրագույն արժեք պարունակող գագաթը չի կարող ունենալ ձախ ենթածառ):

Դիցուք **remove(X, T, T1)**-ը պնդում է, ըստ որի՝ **T1** որոնման ծառը ստացվում է **T** որոնման ծառից **X** տարրը հեռացնելու արդյունքում (ենթադրվում է, որ եթե **X**-ը չի հանդիպում է **T** ծառում, ապա **T1**-ը համընկնում է **T**-ի հետ): Հեռացման գործողության անդրադարձ նկարագրին համապատասխանում է **remove** պրեդիկատի հետևյալ սահմանումը.

```
remove(_, nil, nil):- !.
remove(X, bTree(L, Y, R), bTree(L1, Y, R)):-
                                        X<Y, remove(X, L, L1), !.
remove(X, bTree(L, Y, R), bTree(L, Y, R1)):-
                                        X>Y, remove(X, R, R1), !.
remove(X, bTree(nil, X, R), R):- !.
remove(X, bTree(L, X, nil), L):- !.
remove(X, bTree(L, X, R), bTree(L, Y, R1)):-
                                    min(R, Y), remove(Y, R, R1).
```

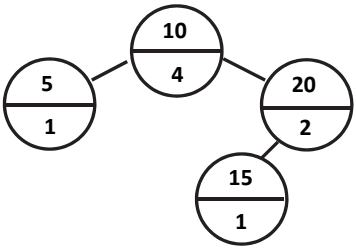
Այստեղ **min(T, X)**-ը պնդում է, ըստ որի՝ **X**-ը **T** ոչ դատարկ որոնման ծառի մինիմալ տարրն է: **min** պրեդիկատի սահմանումն է.

```
min(bTree(nil, X, _), X):-!.
min(bTree(L, _, _), Y):-min(L, Y).
```

10.3. Որոնման ընդլայնված բինար ծառեր

Որոշ դեպքերում լրացուցիչ տեղեկատվության պահպանումը տվյալների կառուցվածքի հանգույցներում հնարավորություն է տալիս արդյունավետ կերպով կատարելու նոր գործողություններ: Այս իմաստով ընդլայնենք որոնման ծառում պահպանվող տեղեկատվությունը՝ նպատակ ունենալով բացի վերը դիտարկված երեք հիմնական գործողություններից արդյունավետ կերպով կատարելու ևս երկու գործողություն, այն է՝ *տարրի կարգահամարի* և *կարգահամարով տարրի* որոշման գործողությունները:

Ենթադրենք, որ որոնման ծառի յուրաքանչյուր գագաթում, արժեքից բացի, գրառվում է հաշվիչ, որի արժեքն է այդ գագաթով որոշվող ենթածառի ծավալը: Որոնման ընդլայնված բինար ծառ է **Նկ. 10.2** պատկերված ծառը:



Նկ. 10.2

Որոնման ընդլայնված բինար ծառի գագաթները պայմանավորվենք նշել

bTree(L, info(Value, Count), R)

կառուցվածքի միջոցով, որտեղ

- **L**-ը և **R**-ը գագաթի ձախ և աջ ենթածառերն են,
- **Value** -ն և **Count**-ը համապատասխանաբար գագաթի և հաշվիչի արժեքներն են:

Օրինակ՝ **Նկ. 10.2**-ում պատկերված ծառին համապատասխանում է հետևյալ կառուցվածքը.

```
bTree(    bTree(nil, info(5, 1), nil),
          info(10, 4),
          bTree(  bTree(nil, info(15, 1), nil),
                  info(20, 2),
                  nil
          )
):
```

10.3.1. Տարրի կարգահամարի որոշում

Դիցուք **X** տարրը հանդիպում է առանց կրկնումների **T** որոնման ընդլայնված ծառում: **X**-ի կարգահամարը **T**-ում սահմանենք որպես **X**-ի կարգահամար **T** ծառի սիմետրիկ կարգով շրջանցման արդյուն-

քում ստացված հաջորդականությունում: Դիտարկենք **X**-ի կարգահամարը **T** ծառում որոշելու հետևյալ անդրադարձ ալգորիթմը.

- եթե **T**-ի ձախ ենթածառը դատարկ է, ապա.
  - եթե **X**-ը **T**-ի արմատի արժեքն է, ապա **X**-ի կարգահամարը **T**-ում **1**-է,
  - հակառակ դեպքում **X**-ի կարգահամարը **T**-ում մեկով մեծ է **X**-ի կարգահամարից **T**-ի աջ ենթածառում,
- եթե **T**-ի ձախ ենթածառը դատարկ չէ, ապա.
  - եթե **X**-ը փոքր է **T**-ի արմատի արժեքից, ապա **X**-ի կարգահամարը **T**-ում **X**-ի կարգահամարն է **T**-ի ձախ ենթածառում,
  - եթե **X**-ը **T**-ի արմատի արժեքն է, ապա **X**-ի կարգահամարը **T**-ում մեկով մեծ է **T**-ի ձախ ենթածառի ծավալից,
  - եթե **X**-ը մեծ է **T**-ի արմատի արժեքից, ապա **X**-ի կարգահամարը **T**-ում մեկով և **T**-ի ձախ ենթածառի ծավալով մեծ է **X**-ի կարգահամարից **T**-ի աջ ենթածառում:

Դիցուք **getOrder(T, X, I)**-ը պնդում է, ըստ որի՝ առանց կրկնումների **T** որոնման ընդլայնված ծառում **X**-ի կարգահամարն է **I**-ն: Տարբի կարգահամարի որոշման անդրադարձ ալգորիթմին համապատասխանում է **getOrder** պրեդիկատի հետևյալ սահմանումը.

```
getOrder(bTree(nil, info(X,_), _), X, 1):- !.
getOrder(bTree(nil, _, R), X, I):- getOrder(R, X, I1), I is I1+1, !.
getOrder(bTree(L, info(Y, _), _), X, I):- X<Y, getOrder(L, X, I),!.
getOrder(bTree(bTree(_, info(_,K1),_), info(X,_),_), X, I):-
                                                    I is K1+1, !.
getOrder(bTree(bTree(_, info(_,K1), _), _, R), X, I):-
                                        getOrder(R, X, I1), I is K1+1+I1.
```

10.3.2. Կարգահամարով տարրի որոշում

Դիցուք **T** որոնման ընդլայնված ծառը պարունակում է առնվազն **I** գագաթ: Դիտարկենք **T** ծառում **I** կարգահամար ունեցող տարրի որոշման հետևյալ անդրադարձ ալգորիթմը.

- եթե **T**-ի ձախ ենթածառը դատարկ է, ապա.
  - եթե **I=1**, ապա **I** կարգահամար ունեցող տարրն է **T**-ում **T-**ի արմատում գրառված տարրը,
  - եթե **I>1**, ապա **I** կարգահամար ունեցող տարրն է **T**-ում **I-1** կարգահամար ունեցող տարրը **T**-ի աջ ենթածառում,
- եթե **T**-ի ձախ ենթածառը դատարկ չէ, ապա.
  - եթե **I**-ն փոքր է կամ հավասար **T**-ի ձախ ենթածառի ծավալից, ապա **I** կարգահամար ունեցող տարրն է **T**-ում **I** կարգահամար ունեցող տարրը **T**-ի ձախ ենթածառում,
  - եթե **I**-ն մեկով մեծ է **T**-ի ձախ ենթածառի ծավալից, ապա **I** կարգահամար ունեցող տարրն է **T**-ում **T**-ի արմատում գրառված տարրը,
  - եթե **I**-ն առնվազն երկուսով մեծ է **T**-ի ձախ ենթածառի ծավալից, ապա **I** կարգահամար ունեցող տարրն է **T**-ում **I**-ից մեկով, ինչպես նաև **T**-ի ձախ ենթածառի ծավալով փոքր կարգահամար ունեցող տարրը **T**-ի աջ ենթածառում:

Դիցուք **getElement(T, I, X)**-ը պնդում է, ըստ որի՝ **T** որոնման ընդլայնված ծառում **I**-կարգահամար ունեցող տարրն է **X**-ը: Կարգահամարով տարրի որոշման անդրադարձ ալգորիթմին համապատասխանում է **getElement** պրեդիկատի հետևյալ սահմանումը.

```
getElement(bTree(nil, info(X,_), _), 1, X):- !.
getElement(bTree(nil, info(_,_), R), I, X):-
                                    I1 is I-1, getElement(R, I1, X),!.
getElement(bTree(bTree(L1, info(X1,K1), R1), _, _), I, X):-
            I=<K1, getElement(bTree(L1, info(X1,K1), R1), I, X), !.
```

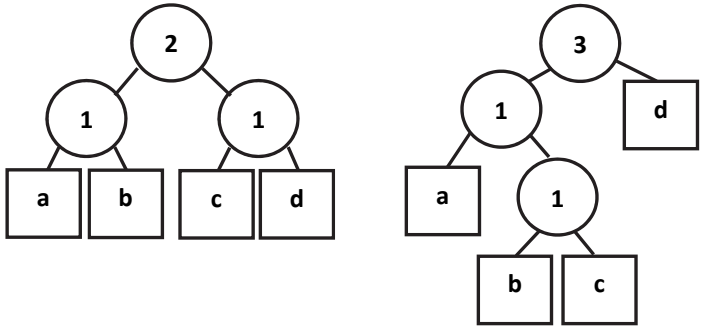
```
getElement(bTree(bTree(_,info(_,K1),_),info(X,_),_), I, X):-
                                                    I is K1+1, !.
getElement(bTree(bTree(_, info(_,K1), _), _, R), I, X):-
                                    I1 is I-K1-1, getElement(R, I1, X).
```

Խնդիրներ

Սահմանում 1: Բինար ծառը կանվանենք *խիստ բինար ծառ*, եթե դրա յուրաքանչյուր ներքին գագաթ ունի ճիշտ երկու զավակ:

Սահմանում 2: *Յուցակի որոնման ծառ* կանվանենք ցանկացած խիստ բինար ծառ, որի տերևները արժևորված են ցուցակի տարրերով, իսկ ներքին գագաթները՝ ամբողջ թվերով այնպես, որ տերևների արժեքների հաջորդականությունը ձախից աջ վերցված համընկնում է ցուցակին, իսկ յուրաքանչյուր ներքին գագաթի արժեքն է այդ գագաթի ձախ ենթածառի տերևների քանակը:

Օրինակ՝ **Նկ. 10.3**-ում պատկերված որոնման ծառերը ներկայացնում են **[a, b, c, d]** ցուցակը.



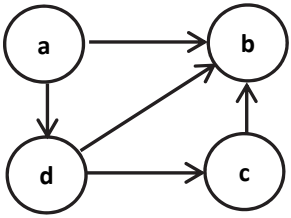
Նկ. 10.3

1. Տրված են որոնման ծառով ներկայացված **L** ցուցակը և **K** ամբողջ թիվը, որտեղ **1≤K≤|L|**: Որոշել **L** ցուցակի **K**-րդ տարրը:

2. Տրված են որոնման ծառով ներկայացված **L** ցուցակը, **X** տարրը և **K** ամբողջ թիվը, որտեղ **1≤K≤|L|+1**: Ներդնել **X**-ը **L** ցուցակի **K**-րդ դիրքում (**K=|L|+1** դեպքում **X**-ը գրառել **L** ցուցակի վերջում): Գործողության կատարման արդյունքում կրկին պետք է ստացվի ցուցակի որոնման ծառ:
3. Տրված են որոնման ծառի տեսքով ներկայացված **L** ցուցակը և **K** ամբողջ թիվը, որտեղ **1≤K≤|L|**: Հեռացնել **L** ցուցակի **K**-րդ տարրը: Գործողության կատարման արդյունքում կրկին պետք է ստացվի ցուցակի որոնման ծառ:

11. Գրաֆներ:
Գրաֆի ներկայացման եղանակներ

Գրաֆը սահմանվում է որպես **G=(V, E)** զույգ, որտեղ **V**-ն և **E**-ն գագաթների և կողերի վերջավոր բազմություններ են: Գրաֆը կոչվում է *ուղղորդված*, եթե կողերը գագաթների կարգավորված զույգեր են և հակառակ դեպքում՝ *չուղղորդված*: Ստորև բերված է ուղղորդված գրաֆի օրինակ:



Նկ. 11.1

11.1. Ներկայացման ձևեր

11.1.1. Ներկայացման առաջին ձև

Գրաֆը կարելի է ներկայացնել՝ թվարկելով դրա գագաթներն ու կողերը փաստերի միջոցով: Ենթադրելով, որ գագաթը ներկայացվում է մեկ տեղանի **vertex**, իսկ կողը՝ երկտեղանի **edge** պրեդիկատի միջոցով, **Նկ. 11.1**-ում պատկերված գրաֆը կարող ենք ներկայացնել հետևյալ կերպ.

```
vertex(a).
vertex(b).
vertex(c).
vertex(d).
```

```
edge(a, b).
edge(a, d).
edge(c, b).
edge(d, b).
edge(d, c).
```

Սահմանենք **link** օժանդակ պրեդիկատ, որն ուղղորդված գրաֆի դեպքում նույնանում է **edge** պրեդիկատի հետ՝

```
link(X, Y):-edge(X, Y).
```

իսկ չուղղորդված գրաֆի դեպքում սահմանվում է որպես գագաթները կապող կող՝ անկախ այն բանից, թե ինչ հերթականությամբ են գագաթները նշված կողում.

```
link(X, Y):-edge(X, Y); edge(Y, X).
```

(այստեղ օգտագործված է կետ-ստորակետ նշանը, որը **Prolog** լեզվում նշում է դիզյունկցիայի գործողությունը):

11.1.2. Ներկայացման երկրորդ ձև

Գրաֆը կարելի է ներկայացնել **graph** բինար պրեդիկատի միջոցով, որի արգումենտներն են գագաթների և կողերի ցուցակները: Այսպես, օրինակ, **Նկ.11.1**-ում պատկերված գրաֆը կներկայացվի հետևյալ փաստի միջոցով.

```
graph(  [a, b, c, d],
        [edge(a,b), edge(a,d), edge(c,b), edge(d,b), edge(d,c)]
).
```

11.1.3. Ներկայացման երրորդ ձև

Վերջապես գրաֆը կարելի է ներկայացնել՝ յուրաքանչյուր գագաթի համար նշելով դրան կից գագաթների ցուցակը: Այս ներկայաց-

ման դեպքում օգտագործենք **graph** ունար պրեդիկատ, որի արգումենտն է հետևյալ տեսք ունեցող կառուցվածքների ցուցակը.

-> (գագաթ, կից գագաթների ցուցակ):

Դիտարկվող կառուցվածքները ստորև կներկայացնենք միջածանցային ձևով, այն է՝

(գագաթ -> կից գագաթների ցուցակ):

Այսպես, օրինակ, **Նկ. 11.1**-ում պատկերված գրաֆը կներկայացվի հետևյալ կերպ.

```
graph(    [ (a->[b, d]),
            (b->[]) ,
            (c->[b]),
            (d->[b, c]) ]
).
```

Նկատենք, որ չուղղորդված գրաֆի յուրաքանչյուր **(X, Y)** կողի համար **Y** գագաթը պետք է հանդիպի **X** գագաթին կից գագաթների ցուցակում, իսկ **X** գագաթը՝ **Y** գագաթին կից գագաթների ցուցակում:

11.2. Անցում ներկայացման առաջին ձևից ներկայացման այլ ձևերին

Գրաֆն ի սկզբանե հարմար է ներկայացնել առաջին ձևով՝ անդրադարձ մշակում կատարելու անհրաժեշտության դեպքում անցնելով ներկայացման այլ ձևերին: Ստորև դիտարկենք համապատասխան ձևափոխությունների իրականացումը:

Գագաթների ցուցակի կառուցում: Գրաֆի գագաթների ցուցակը կարելի է ստանալ գագաթները սահմանող փաստերի հիման վրա՝ մաքսիմալ չափով ընդլայնելով գագաթների դատարկ ցուցակը.

```
vertices(V):-extend([], V).
```

Այստեղ **vertices(V)**-ն պնդում է, ըստ որի՝ **V**-ն առաջին ձևով ներկայացված գրաֆի գագաթների ցուցակն է, իսկ **extend([], V)**-ն՝ պնդում այն մասին, որ գագաթների **V** ցուցակը ստացվում է գագաթների դատարկ ցուցակի մաքսիմալ ընդլայնման արդյունքում: Ընդհանուր դեպքում նշանակենք **extend(V1, V)**-ով պնդումը, ըստ որի՝ գագաթների **V** ցուցակը ստացվում է գագաթների **V1** ցուցակի մաքսիմալ ընդլայնման արդյունքում՝ գագաթները թվարկող փաստերի հիման վրա: **extend** պրեդիկատը սահմանենք հետևյալ կերպ.

```
extend(V1, V):- addNewVertex(V1, V2), extend(V2, V), !.
extend(V, V).
```

որտեղ **addNewVertex(V1, V2)**-ը պնդում է, ըստ որի՝ գագաթների **V2** ցուցակը ստացվում է գագաթների **V1** ցուցակին նոր գագաթ ավելացնելու արդյունքում: **addNewVertex** պրեդիկատի սահմանումն է.

```
addNewVertex(V, [X|V]):- vertex(X), notMember(X, V).

notMember(X, L):- member(X, L),!, fail.
notMember(_, _).
```

Կողերի ցուցակի կառուցում: Կողերի ցուցակը կարելի է կառուցել այնպես, ինչպես կառուցվեց գագաթների ցուցակը: Ստորև բերված ծրագրում ենթադրվում է, որ.

- **edges(E)**-ն պնդում է, ըստ որի՝ **E**-ն առաջին ձևով ներկայացված գրաֆի կողերի ցուցակն է,
- **extend1(E1, E)**-ն պնդում է, ըստ որի՝ կողերի **E** ցուցակը ստացվում է կողերի **E1** ցուցակի մաքսիմալ ընդլայնման արդյունքում,
- **addNewEdge(E1, E2)**-ը պնդում է, ըստ որի՝ կողերի **E2** ցուցակը ստացվում է կողերի **E1** ցուցակին նոր կող ավելացնելու արդյունքում:

```
edges(E):-          extend1([], E).

extend1(E1, E):-  addNewEdge(E1, E2), extend1(E2, E), !.
extend1(E, E).

addNewEdge(E1,[edge(X, Y)|E1]):-  edge(X, Y),
                                  notMember(edge(X, Y), E1).
```

Կից գագաթների ցուցակի կառուցում: Տրված գագաթին կից գագաթների ցուցակը կարելի է կառուցել՝ ընտրելով դրան կից գագաթները բոլոր գագաթների ցուցակից:

Նշանակենք **adjVertices(A, H)**-ով պնդումը, ըստ որի՝ **H**-ը **A** գագաթին կից գագաթների ցուցակն է: **adjVertices** պրեդիկատը սահմանենք հետևյալ կերպ.

```
adjVertices(A, H):-  vertices(V), adjVertices1(A, V, H).
```

որտեղ **adjVertices1(A, V, H)**-ը պնդում է, ըստ որի՝ **H**-ը գագաթների **V** ցուցակին պատկանող **A**-ին կից գագաթների ցուցակն է: **adjVertices1** պրեդիկատը սահմանենք հետևյալ կերպ.

```
adjVertices1(_, [], []):-  !.
adjVertices1(A, [X|V], [X|H]):-  link(A, X),
adjVertices1(A, V, H), !.
adjVertices1(A, [_|V], H):-  adjVertices1(A, V, H).
```

11.2.1. Անցում ներկայացման երկրորդ ձևին

Օգտագործելով **vertices** և **edges** պրեդիկատները՝ անցումը ներկայացման առաջին ձևից ներկայացման երկրորդ ձևին կարող ենք կատարել հետևյալ կերպ.

```
graph(V, E):-vertices(V), edges(E).
```

11.2.2. Անցում ներկայացման երրորդ ձևին

Օգտագործելով **adjVertices** պրեդիկատը՝ անցումը ներկայացման առաջին ձևից ներկայացման երրորդ ձևին կարող ենք կատարել հետևյալ կերպ.

```
graph(W):- vertices(V), graph1(V, W).

graph1([],[]):-!.
graph1([A|V],[(A->H)|W]):- adjVertices(A, H), graph1(V, W).
```

12. Գրաֆների հետ կապված խնդիրներ

12.1. Երկու գագաթների միջև ճանապարհի գոյություն

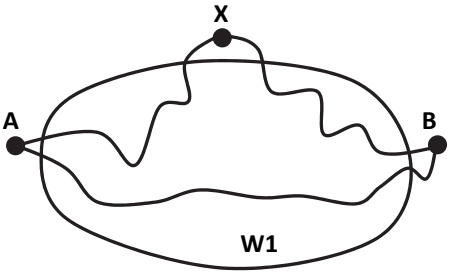
Ճանապարհը գրաֆում սահմանվում է որպես գագաթների այնպիսի հաջորդականություն, որում յուրաքանչյուր երկու հարևան գագաթ կազմում են կող (միացված են կողով):

Դիտարկենք երկու գագաթների միջև ճանապարհի գոյության խնդիրը՝ հիմք ընդունելով *Ֆլոյդի-Վարշալի* ալգորիթմը: Ալգորիթմի հիմքում ընկած է դիտարկվող խնդրի հետևյալ ընդհանրացումը.

Դիցուք **G=(V, E)** գրաֆում տրված են **A** և **B** գագաթները և գագաթների **W ⊆ V \ {A, B}** բազմությունը: Անհրաժեշտ է որոշել, թե գոյություն ունի արդյոք **A** գագաթից **B** գագաթ տանող այնպիսի ճանապարհ, որում, **A** և **B** գագաթներից բացի, մնացած բոլոր գագաթները պատկանում են **W** բազմությանը: Նկատենք, որ **W=V \ {A, B}** դեպքում այս խնդիրը համընկնում է **A** գագաթից **B** գագաթ տանող ճանապարհի գոյության խնդրին:

Ենթադրենք, որ **A≠B** և դիտարկենք **W** բազմության միջոցով **A** և **B** գագաթները միմյանց հետ կապող ճանապարհի գոյությունը որոշելու հետևյալ անդրադարձ ալգորիթմը.

- եթե **W=∅**, ապա արտահանել **“այո” ⇔ (A, B)∈E**,
- եթե **W≠∅**, ապա դիցուք **X**-ը **W**-ին պատկանող որևէ գագաթ է, **W1=W\{X}**: Նկատենք, որ **W** բազմության միջոցով **A** և **B** գագաթները միմյանց հետ կապող ճանապարհ գոյություն ունի այն և միայն այն դեպքում, երբ.
  - գոյություն ունի **W1** բազմության միջոցով **A** և **B** գագաթները միմյանց հետ կապող ճանապարհ, կամ
  - գոյություն ունեն **W1** բազմության միջոցով **A** և **X**, ինչպես նաև **X** և **B** գագաթները միմյանց հետ կապող ճանապարհներ.



Նկ. 12.1

Նշանակենք **path(A, B)** պնդումը, ըստ որի՝ գոյություն ունի **A** և **B** գագաթները միմյանց հետ կապող ճանապարհ առաջին ձևով ներկայացված **G** գրաֆում: **path** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
path(A, A):-!.
path(A, B):- vertices(V), remove(A, V, V1), remove(B, V1, V2),
                                                path1(A, B, V2).
```

Այստեղ **path1(A, B, W)**-ն պնդում է, ըստ որի՝ գոյություն ունի գագաթների **W** ցուցակի (բազմության) միջոցով **A** և **B** գագաթները միմյանց հետ կապող ճանապարհ **G** գրաֆում: Հիմք ընդունելով վերը արված դիտարկումը՝ (տես **Նկ. 12.1**) **path1** պրեդիկատը սահմանենք հետևյալ կերպ.

```
path1(A, B, []):-     link(A, B), !.
path1(A, B, [_|W]):- path1(A, B, W), !.
path1(A, B, [X|W]):- path1(A, X, W), path1(X, B, W).
```

remove(A, V, V1)- ը պնդում է, ըստ որի՝ **V1** ցուցակը ստացվում է **V** ցուցակից **A** տարրը հեռացնելու արդյունքում: Այս պրեդիկատի սահմանումն է.

```
remove(A, [A|V], V):- !.
remove(A, [X|V], [X|V1]):-remove(A, V, V1).
```

12.2. Դինամիկ ծրագրավորման մեթոդի կիրառում

Երկու գագաթների միջև ճանապարհի գոյության որոշման վերը դիտարկված անդրադարձ ալգորիթմը կարող է բերել բազմիցս միևնույն ենթախնդիրների լուծման: Ավելի արդյունավետ ալգորիթմ կառուցելու համար կիրառենք *վերից վար* դինամիկ ծրագրավորման մեթոդը, որի հիմքում ընկած է հետևյալ գաղափարը.

- ներմուծվում է *բառարան* տվյալների կառուցվածք, որտեղ բանալու դերում հանդես է գալիս ենթախնդիրը, իսկ բանալուն կից տվյալի դերում՝ ենթախնդրի լուծումը,
- ոչ տրիվիալ խնդիր լուծելիս ենթախնդրի համար անդրադարձ կանչ կատարելուց առաջ նախ ստուգվում է, թե առկա է արդյոք բառարանում այդ ենթախնդրի լուծումը: Եթե ոչ, ապա կատարվում է անդրադարձ կանչը, այլապես բառարանից արտահանվում է այնտեղ գտնվող լուծումը, և կանչը չի կատարվում,
- ցանկացած (տրիվիալ կամ ոչ տրիվիալ) խնդրի լուծումը վերադարձնելուց առաջ այն պահպանվում է բառարանում:

Վերադառնալով մեր խնդրին՝ ենթադրենք, որ գրաֆի գագաթները ամբողջ թվեր են, և հետևաբար ենթախնդիրները (այն է՝ գոյություն ունե՞ն արդյոք տրված երկու գագաթները միմյանց հետ կապող ճանապարհներ, թե՞ ոչ) բառարանագրական կարգով կարգավորված ամբողջ թվերի զույգեր են: Պարզության համար բառարանում պահենք միայն դրական լուծում ունեցող ենթախնդիրներ: Այս դեպքում բառարանի փոխարեն բավարար է օգտագործել դրական լուծում ունեցող ենթախնդիրների (ամբողջ թվերի զույգերի) պահոց, որի դերում կարող է հանդես գալ, օրինակ՝ ամբողջ թվերի զույգերի որոնման ծառը: Արդյունքում որոշելու համար՝ նախկինում հաստատվե՞լ է արդյոք **A** և **B** գագաթները միմյանց հետ կապող ճանապարհի գոյությունը, թե՞ ոչ, բավարար է պարզել՝ պատկանու՞մ է արդյոք **(A, B)** զույգը դիտարկվող որոնման ծառին, թե՞ ոչ:

Բերենք ամբողջ թվերի զույգերի որոնման ծառի համար տարրի ավելացման և տարրի որոնման գործողությունների իրականացումները՝ ենթադրելով, որ տարրերը ներկայացված են **pair(A, B)** տեսքի կառուցվածքների միջոցով.

```
less(pair(X, _), pair(X1, _)):- X<X1, !.
less(pair(X, Y), pair(X, Y1)):- Y<Y1.

search(Pair, bTree(_,Pair,_)):- !.
search(Pair, bTree(L, Pair1, _)):-
                              less(Pair, Pair1), search(Pair, L), !.
search(Pair, bTree(_, _, R)):- search(Pair, R).

insert(Pair, nil, bTree(nil, Pair, nil)):- !.
insert(Pair, bTree(L, Pair, R), bTree(L, Pair, R)):- !.
insert(Pair, bTree(L, Pair1, R), bTree(L1, Pair1, R)):-
                               less(Pair, Pair1), insert(Pair, L, L1),!.
insert(Pair, bTree(L, Pair1, R), bTree(L, Pair1, R1)):-
                                                  insert(Pair, R, R1).
```

Նշենք, որ ավելացման գործողության իրականացումից հետևում է, որ դրական լուծում ունեցող ենթախնդիրների պահոցում (ծառում) չեն կարող լինել կրկնվող ենթախնդիրներ: Այժմ ձևափոխենք նախորդ բաժնում կառուցված ծրագիրը դինամիկ ծրագրավորման մեթոդով կառուցված ծրագրի:

Նշանակենք **dpPath(A, B, M)** պնդումը, ըստ որի՝ **(A, B)** խնդիրը ունի դրական լուծում (այն է՝ գոյություն ունի **A** գագաթից **B** գագաթ տանող ճանապարհ), իսկ **M**-ը **(A, B)** խնդրի լուծման ընթացքում առաջացած դրական լուծում ունեցող ենթախնդիրների որոնման ծառն է: Նկատենք, որ.

```
path(A, B):- dpPath(A, B, _).
```

dpPath պրեդիկատը սահմանենք հետևյալ կերպ.

```
dpPath(A, A, nil):- !.
```

```
dpPath(A, B, M):- vertices(V),
                  remove(A, V, V1), remove(B, V1, V2),
                  dpPath1(A, B, V2, nil, M).
```

dpPath1(A, B, W, M, M1)-ը պնդում է, ըստ որի՝ գոյություն ունի **A** գագաթից **B** գագաթ տանող այնպիսի ճանապարհ, որի միջանկյալ գագաթները պատկանում են գագաթների **W** ցուցակին, և այդ լուծումը ձևափոխում է դրական լուծում ունեցող ենթախնդիրների **M** որոնման ծառը **M1** որոնման ծառին: **dpPath1** պրեդիկատի սահմանումն է.

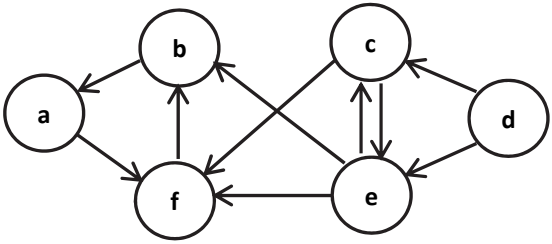
```
dpPath1(A, B, [], M, M1):- link(A, B), insert(pair(A, B), M, M1), !.
dpPath1(A, B, [_|W], M, M1):-
  ( search(pair(A, B), M), M2=M; dpPath1(A, B, W, M, M2) ),
                                 insert(pair(A, B), M2, M1), !.

dpPath1(A, B, [X|W], M, M1):-
  ( search(pair(A, X), M), M2=M; dpPath1(A, X, W, M, M2) ),
  ( search(pair(X, B), M2), M3=M2; dpPath1(X, B, W, M2, M3)),
                                 insert(pair(A, B), M3, M1).
```

12.3. Կապակցվածության կոմպոնենտների կառուցում

Ուղղորդված գրաֆի երկու գագաթ անվանենք *համարժեք*, եթե գրաֆում առկա են ինչպես առաջին գագաթից դեպի երկրորդ գագաթ տանող, այնպես էլ երկրորդ գագաթից դեպի առաջին գագաթ տանող ճանապարհներ (նկատենք, որ չուղղորդված գրաֆի դեպքում բավարար է պահանջել, որ լինի միայն առաջին գագաթից դեպի երկրորդ գագաթ տանող ճանապարհ): Հեշտ է տեսնել, որ սահմանված հարաբերությունը համարժեքության հարաբերություն է գրաֆի գագաթների բազմության վրա, ուստի այն տրոհում է գագաթների բազմությունը *կապակցվածության կոմպոնենտներ* կոչվող համարժեքության դասերի: Այսպես, օրինակ, **Նկ. 12.2**-ում բերված ուղղորդ-

ված գրաֆի կապակցվածության կոմպոնենտներն են **{a, b, f}, {c, e}** և **{d}** գագաթների բազմությունները:



Նկ. 12.2

Դիտարկենք **G=(V, E)** գրաֆի կապակցվածության կոմպոնենտների նկարագրման հետևյալ եղանակը.

- եթե V=∅, ապա կապակցվածության կոմպոնենտների բազմությունը դատարկ է,
- հակառակ դեպքում, եթե.
  - **A∈V**,
  - **C**–ն **A**-ն պարունակող կապակցվածության կոմպոնենտն է, **V1=V\C**,
  - **CC**-ն **V1** բազմության տրոհումն է կապակցվածության կոմպոնենտների,

ապա.

G գրաֆի կապակցվածության կոմպոնենտների բազմությունն է **{C}** ∪ **CC** -ն:

Նշանակենք **connectedComponents(CC)**-ով պնդումը, ըստ որի՝ **CC**-ն առաջին ձևով ներկայացված ուղղորդված գրաֆի կապակցվածության կոմպոնենտների ցուցակն է: **connectedComponents** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
connectedComponents(CC):- vertices(V),
                          connectedComponents1(V, CC).
```

որտեղ **connectedComponents1** պրեդիկատի սահմանումն է.

```
connectedComponents1([], []):-!.
connectedComponents1([A|V], [C|CC]):-
                                component(A, [A|V], C, V1),
                                connectedComponents1(V1, CC).
```

Վերջապես, **component(A, V, C, V1)**-ը պնդում է, ըստ որի՝ **C**-ցուցակը կազմված է **V** ցուցակի **A**-ին համարժեք տարրերից, իսկ **V1** ցուցակը՝ **V** ցուցակի մնացած տարրերից: Ստորև բերված է **component** պրեդիկատի սահմանումը.

```
component(_, [], [], []):- !.
component(A, [X|V], [X|C], V1):- path(A, X), path(X, A),
                                 component(A, V, C, V1), !.
component(A, [X|V], C, [X|V1]):- component(A, V, C, V1).
```

Խնդիրներ

1. Չուղղորդված գրաֆի ճանապահի երկարությունը սահմանենք որպես կողերի քանակ, իսկ երկու գագաթների միջև եղած հեռավորությունը՝ որպես այդ գագաթները միացնող կարճագույն ճանապարհի երկարություն (ենթադրվում է, որ գագաթների միջև ընկած հեռավորությունը անվերջ է, եթե այդ գագաթները միացնող ճանապարհ գոյություն չունի): Չնափոխել **12.1** բաժնում բերված ճանապարհի գոյության որոշման ալգորիթմն այնպես, որ հնարավոր լինի գտնել երկու գագաթների միջև եղած հեռավորությունը:
2. Օգտագործելով երկու գագաթների միջև հեռավորության որոշման ծրագիրը՝ գտնել չուղղորդված կապակցված գրաֆի տրամագիծը, այն է՝ մաքսիմալ հեռավորությունը երկու գագաթների միջև:

13. Վերջավոր ավտոմատներ և քերականություններ

13.1. Վերջավոր ավտոմատներ

13.1.1. Վերջավոր ավտոմատի սահմանում

Վերջավոր ավտոմատը սահմանվում է որպես **M=(Q, Σ, δ, q0, F)** հնգյակ, որտեղ

- **Q**-ն վիճակների վերջավոր բազմությունն է,
- **Σ**-ն մուտքային սիմվոլների վերջավոր բազմությունն է,
- **δ: Q×Σ→Q** անցումների ֆունկցիան է,
- **q0∈Q** սկզբնական վիճակն է,
- **F⊆Q** վերջնական վիճակների բազմությունն է:

M վերջավոր ավտոմատը ներկայացնենք **G(M)** ուղղորդված նիշավորված գրաֆի միջոցով, որի գագաթների բազմությունն է **Q**-ն, և որում **q∈Q** գագաթը միացված է **a**-ով նշված սլաքով **q'∈Q** գագաթի հետ, եթե **δ(q, a)=q'**: Գրաֆով ներկայացումը ամբողջական դարձնելու համար պայմանավորվենք մուտքային վիճակը նշել մտնող սլաքով, իսկ ելքային վիճակները՝ վերցնել կրկնակի շրջանագծի մեջ: Օրինակ՝ **M=(Q, Σ, δ, q0, F)** ավտոմատին, որտեղ

Q={q0, q1, q2}, Σ={a, b},

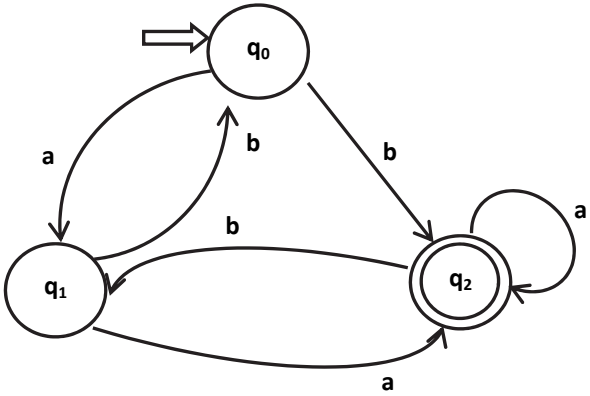
δ={(q0,a)->q1,(q0,b)->q2,

(q1,a)->q2,(q1,b)->q0,

(q2,a)->q2,(q2,b)->q1},

q0, F={q2}

համապատասխանում է **G(M)** հետևյալ նիշավորված գրաֆը (գծապատկերը).



Նկ. 13.1

Եթե

$$h = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} q_k$$ -ը

ճանապարհ է **G(M)** գրաֆում, ապա $w=a_1a_2...a_k$ բառը կանվանենք ***h*** *ճանապարհով ծնվող բառ*: Կասենք, որ ***w*** *բառը ճանաչվում է* ***M*** *ավտոմատով*, եթե սկզբնական վիճակից սկսվող և **w** բառը ծնող (միակ) ճանապարհը **G(M)** գրաֆում ավարտվում է վերջնական վիճակում:

M *ավտոմատով ճանաչվող* ***L(M)*** *լեզուն* սահմանենք որպես **M** ավտոմատով ճանաչվող բոլոր բառերի բազմություն: Օրինակ, **aa, abb, aabbb** բառերը պատկանում են **Նկ. 13.1**–ում բերված ավտոմատով ճանաչվող լեզվին, իսկ **ab** և **bab** բառերը՝ ոչ:

13.1.2. Վերջավոր ավտոմատի ներկայացում Prolog լեզվով

M վերջավոր ավտոմատը **Prolog** լեզվով ներկայացնելու համար օգտագործենք հետևյալ պրեդիկատները.

- **states**
  /*states(Q) ⇔ Q-ն ավտոմատի վիճակների ցուցակն է***/**:
- **symbols**
  /*symbols(Sigma) ⇔ Sigma-ն ավտոմատի մուտքային սիմվոլների ցուցակն է***/**:
- **transition**
  /*transition(X, A, Y) ⇔δ (X, A)=Y*/:
- **startState**
  /*startState(S) ⇔ S-ը ավտոմատի սկզբնական վիճակն է***/**:
- **finalStates**
  /*finalStates(F) ⇔ F-ը ավտոմատի վերջնական վիճակների ցուցակն է***/**:

Այսպես, օրինակ, **Նկ. 13.1**-ում պատկերված վերջավոր ավտոմատը կներկայացվի հետևյալ փաստերի միջոցով.

```
states([q0, q1, q2]).

symbols([a, b]).

transition(q0, a, q1).
transition(q0, b, q2).
transition(q1, a, q2).
transition(q1, b, q0).
transition(q2, a, q1).
transition(q2, b, q2).

startState(q0).

finalStates([q2]).
```

13.1.3. Բառի ճանաչում

Դիցուք տրված է **w** բառը մուտքային սիմվոլների այբուբենում և անհրաժեշտ է որոշել՝ պատկանում է արդյոք այն ավտոմատով ճանաչվող լեզվին, թե ոչ: Այս խնդրի փոխարեն դիտարկենք ավելի ընդհանուր խնդիր:

Տրված են **w** բառը մուտքային այբուբենում և ավտոմատին պատկանող **q** վիճակը: Պահանջվում է որոշել՝ պատկանում է արդյոք **w** բառը ավտոմատի **q** վիճակից ճանաչվող լեզվին, թե ոչ (կամ, այլ կերպ ասած, վերջանու՞մ է արդյոք ավտոմատի **q** վիճակից սկսվող և **w** բառը ծնող ճանապարհը վերջնական վիճակում, թե՞ ոչ):

Նկատենք, որ **w**-ն պատկանում է ավտոմատի **q** վիճակից ճանաչվող լեզվին այն և միայն այն դեպքում, երբ.

- **w**-ն դատարկ է, իսկ **q**-ն՝ վերջնական վիճակ,
- **w=aw$_1$**, որտեղ **a**-ն պատկանում է **Σ**-ին, իսկ **w$_1$**-ը՝ ավտոմատի **q$_1$=δ(q, a)** վիճակից ճանաչվող լեզվին:

Նշանակենք **accepted(W)** պնդումը, ըստ որի՝ ավտոմատը ճանաչում է **W** ցուցակով ներկայացված բառը: **accepted** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
accepted(W):-startState(Q0), accepted1(Q0, W).
```

Այստեղ **accepted1(Q, W)**-ն պնդում է, ըստ որի՝ **W** ցուցակով ներկայացված բառը պատկանում է ավտոմատի **Q** վիճակից ճանաչվող լեզվին: **accepted1** պրեդիկատի սահմանումն է.

```
accepted1(Q, []):-  finalStates(F), !, member(Q, F).
accepted1(Q, [A|W]):-  transition(Q, A, Q1),
accepted1(Q1, W),!.
```

որտեղ **member**-ը ցուցակին տարրի պատկանելիության պրեդիկատն է:

13.2. Կոնտեքստից ազատ քերականություններ

13.2.1. Քերականության սահմանում

Կոնտեքստից ազատ քերականությունը սահմանվում է որպես **G=(N, Σ, P, S)** քառյակ, որտեղ

- **N**-ն օժանդակ սիմվոլների վերջավոր բազմությունն է,
- **Σ**-ն հիմնական սիմվոլների վերջավոր բազմությունն է,
- **P⊆N×(N∪Σ)*** կանոնների վերջավոր բազմությունն է,
- **S∈N** քերականության սկզբնական սիմվոլն է:

(A, α)∈P կանոնը պայմանավորվենք նշանակել **A→α**, իսկ **A→α_1, … , A→α_k** կանոնների հաջորդականությունը՝ **A→α_1| … | A→α_k**:

Օրինակ: Կոնտեքստից ազատ քերականություն է **G_0=(N, Σ, P, S)**-ը, որտեղ

N={E, T, F},
Σ={a, *, + (,)},
P={ E→T| T+E, T→F| F*T, F→a| (E) },
S=E

(**E, T, F** օժանդակ սիմվոլները համապատասխանում են *արտահայտություն* (**Expression**), *թերմ* (**Term**) և *արտադրիչ* (**Factor**) հասկացություններին):

Դիցուք տրված են **G=(N, Σ, P, S)** կոնտեքստից ազատ քերականությունը և **α, β∈(N∪Σ)*** բառերը:

Կասենք, որ ***β*-ն անմիջականորեն արտածվում է *α*-ից**, և կգրենք **α⇒β**, եթե

α=α_1Aα_2, β=$\alpha_1\gamma\alpha_2$, A→γ∈P:

Տրված $k\geq 0$ համար կասենք, որ β-ն ***k** քայլերից հետո արտածվում է α-ից*, և կգրենք $\alpha\Rightarrow^k\beta$, եթե գոյություն ունեն այնպիսի $\gamma_0, \ldots, \gamma_k\in(N\cup\Sigma)^*$ բառեր, որ

$$\gamma_0\Rightarrow\gamma_1, \ldots, \gamma_{k-1}\Rightarrow\gamma_k, \gamma_0=\alpha, \gamma_k=\beta$$

(նշենք, որ β-ն 0 քայլերից հետո արտածվում է α-ից $\Leftrightarrow \alpha=\beta$):

Կասենք, որ *β-ն արտածվում է α-ից* և կգրենք $\alpha\Rightarrow^*\beta$, եթե գոյություն ունի այնպիսի $k\geq 0$, որ $\alpha\Rightarrow^k\beta$:

***G** քերականությամբ ծնվող **L(G)** լեզուն* սահմանենք որպես սկզբնական նշանից արտածվող և հիմնական սիմվոլներից բաղկացած բոլոր բառերի բազմություն.

$$L(G)=\{w\in\Sigma^* \mid S\Rightarrow^* w\}:$$

Օրինակ՝ վերը սահմանված G_0 քերականության համար $L(G_0)$ լեզվին են պատկանում հետևյալ բառերը.

$$((a)),\ a*(a+a),\ (a+(a*a))*a+a:$$

Սահմանում: **G** կոնտեքստից ազատ քերականությունը կանվանենք *առանց ձախ անդրադարձման* քերականություն, եթե դրանում հնարավոր չէ $A\Rightarrow^* A\alpha$ տեսքի արտածում, որտեղ $A\in N, \alpha\in(N\cup\Sigma)^*$:

Ստույգ է հետևյալ պնդումը. ցանկացած կոնտեքստից ազատ քերականության համար գոյություն ունի դրան համարժեք առանց ձախ անդրադարձման կոնտեքստից ազատ քերականություն:

13.2.2. Ներկայացում Prolog լեզվով

G կոնտեքստից ազատ քերականությունը **Prolog** լեզվով ներկայացնելու համար օգտագործենք հետևյալ պրեդիկատները.

- **nonterminals**
  /*nonterminals(Nu) ⇔ Nu-ն քերականության օժանդակ սիմվոլների ցուցակն է ***/**:

- **terminals**
  /*terminals(Sigma) ⇔ **Sigma**-ն քերականության հիմնական սիմվոլների ցուցակն է ***/**:
- **production**
  /*production(A, Alpha) ⇔ քերականությունն ունի **A** օժանդակ սիմվոլով սկսվող այնպիսի կանոն, որի աջ մասի սիմվոլների հաջորդականությունն է **Alpha** ցուցակը ***/**:
- **startSymbol**
  /*startSymbol(S) ⇔ **S**-ը քերականության սկզբնական սիմվոլն է ***/**:

Օրինակ՝ G_0 քերականությունը կներկայացվի հետևյալ փաստերի միջոցով.

```
nonterminals(['E', 'T', 'F']).

terminals([a, +, *, '(', ')']).

production('E', ['T']).
production('E', ['T', +, 'E']).
production('T', ['F']).
production('T', ['F', *, 'T']).
production('F', [a]).
production('F', ['(', 'E', ')']).

startSymbol('E').
```

13.2.3. Բառի արտածելիության ստուգում

Դիցուք տրված է **w** բառը հիմնական սիմվոլների այբուբենում, և անհրաժեշտ է որոշել՝ արտածվում է արդյոք այն տրված քերականությամբ, թե ոչ: Ինչպես դա արեցինք ավտոմատների դեպքում, այստեղ նույնպես դիտարկենք ավելի ընդհանուր խնդիր:

Տրված են **w** բառը հիմնական սիմվոլների այբուբենում և գոնե մեկ օժանդակ սիմվոլ պարունակող α բառը հիմնական և օժանդակ

սիմվոլների այբուբենում: Անհրաժեշտ է ստուգել՝ հավաստի՞ է արդյոք $\alpha \Rightarrow^* w$ պնդումը:

Նշենք, որ $\alpha \Rightarrow^* w$ այն և միայն այն դեպքում, երբ

- α-ն և w-ն դատարկ են, կամ
- $\alpha = a\alpha_1$, $w = aw_1 (a \in T)$ և $\alpha_1 \Rightarrow^* w_1$, կամ
- $\alpha = A\alpha_2$ $(a \in N)$, և գոյություն ունի այնպիսի $A \rightarrow \alpha_1$ կանոն, որ $\alpha_1\alpha_2 \Rightarrow^* w$:

Նշանակենք **derived(W)** պնդումը, ըստ որի՝ **W** ցուցակով ներկայացված բառը արտածվում է առանց ձախ անդրադարձման կոնտեքստից ազատ քերականությամբ: **derived** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
derived(W):- startSymbol(S), derived1([S], W).
```

Այստեղ **derived1(Alpha, W)**–ն պնդում է, ըստ որի՝ հիմնական և օժանդակ սիմվոլներից բաղկացած **Alpha** ցուցակից (**Alpha** ցուցակով ներկայացված բառից) արտածվում է հիմնական սիմվոլներից բաղկացած **W** ցուցակը (**W** ցուցակով ներկայացված բառը): Ստորև բերված է **derived1** պրեդիկատի սահմանումը.

```
derived1([], []):- !.
derived1([X|Alpha], [X|W]):- !, derived1(Alpha, W).
derived1([A|Alpha], W):- production(A, Right),
        append(Right, Alpha, Alpha1), derived1 (Alpha1, W), !.
```

Խնդիրներ

1. Տրված են **M** վերջավոր ավտոմատը և **K** ոչ բացասական ամբողջ թիվը: Որոշել **M** ավտոմատով ճանաչվող **K** երկարություն ունեցող բառերը:

2. Տրված են առանց ձախ անդրադարձման **G** կոնտեքստից ազատ քերականությունը և **K** ոչ բացասական ամբողջ թիվը: Որոշել **G** քերականությամբ արտածվող **K** երկարություն ունեցող բառերը:

14. Լեքսիկական վերլուծություն

14.1. Լեքսիկական վերլուծության խնդիր

Լեքսիկական վերլուծության խնդիրը սահմանենք հետևյալ կերպ. Տրված են.

1. լեքսեմների տիպերը սահմանող վերջավոր ավտոմատների $M_1, \ldots, M_k$ կարգավորված հաջորդականությունը: Ենթադրվում է, որ դիտարկվող ավտոմատները ճանաչում են միայն ոչ դատարկ բառեր,
2. W բառը Σ այբուբենում:

Պահանջվում է կառուցել W բառի ենթաբառերից և ամբողջ թվերից կազմված զույգերի այնպիսի

$$(L_1, J_1) \ \ldots \ (L_m, J_m)$$

հաջորդականություն, որ յուրաքանչյուր s –ի համար $(1 \le s \le m)$ բավարարվեն հետևյալ պայմանները.

- $L_1 \ldots L_{s-1}$-ը W բառի նախածանց է,
- L_s-ը W բառում $L_1 \ldots L_{s-1}$ բառին հաջորդող այն ամենամեծ ենթաբառն է, որը ճանաչվում է տրված ավտոմատներից որևէ մեկով,
- J_s-ը L_s բառը ճանաչող առաջին ավտոմատի կարգահամարն է:

Մենք կենթադրենք, որ եթե W բառի $L_1 \ldots L_{s-1}$ նախածանցին հաջորդող և ավտոմատներից որևէ մեկով ճանաչվող ենթաբառ չկա, ապա L_s-ը W բառի $L_1 \ldots L_{s-1}$ նախածանցին հաջորդող վերջածանցն է, $J_s=-1$, $m=s$:

Օրինակ, եթե լեքսեմների տիպերը տրված են հետևյալ կանոնավոր արտահայտություններով՝

$$R_1=(b+1)^*ab,\ R_2=(b++1)(ab)^*,\ R_3=a(ab^*+0)1,$$

ապա

$$W=aa1bbbab1ababb$$

բառի լեքսիկական վերլուծության արդյունքն է զույգերի

(aa1,3)(bbbab,1)(1abab,2)(b,2)

հաջորդականությունը, իսկ

W=bbbaaa

բառի՝ զույգերի

(bbb,2)(aaa, -1)

հաջորդականությունը:

14.2. Առաջին լեքսեմի որոշում

Դիտարկենք հետևյալ մասնավոր խնդիրը.

Տրված են.

1. դատարկ բառը չճանաչող վերջավոր ավտոմատների $\mathbf{M_1 \ldots, M_k}$ կարգավորված հաջորդականությունը,
2. **W** բառը **Σ** այբուբենում:

Պահանջվում է գտնել ավտոմատներից որևէ մեկի միջոցով ճանաչվող **W** բառի ամենամեծ նախածանցը և դրան ճանաչող ավտոմատի կարգահամարը: Եթե ամենաերկար նախածանցը միաժամանակ ճանաչվում է մեկից ավելի թվով ավտոմատներով, ապա անհրաժեշտ է ընտրել դրանցից առաջինին: Վերջապես, եթե ավտոմատներից որևէ մեկի միջոցով ճանաչվող **W** բառի նախածանց չկա, ապա անհրաժեշտ է արձանագրել այս փաստը:

Ձևակերպված խնդիրը, որը կանվանենք *առաջին լեքսեմի որոշման խնդիր*, լուծենք հետևյալ ալգորիթմով:

Ալգորիթմ T. Առաջին լեքսեմի որոշում:

Մուտք:

- Դատարկ բառը չճանաչող վերջավոր ավտոմատների **M[J]=(Q[J], Σ, δ[J], q0[J], F[J]), 1≤J≤K,** հաջորդականություն,

- **W=A[I], 1≤I≤N (N≥0)** բառ **Σ** այբուբենում:

Ելք:

- **length=max{ I≥0| (∃J)[A[1] … A[I]∈L(M[J])},**
- **index=min{ J| A[1]… A[length]∈L(M[J])}:**
  (Եթե **W** բառը ավտոմատներից որևէ մեկով ճանաչվող նախածանց չունի, ապա **length=N, index=-1**)

Մեթոդ:

T1 [Սկզբնական արժևորում]:
length←N, index←-1,
I←0,
J=1,…, K համար կատարել **q[J]←q0[J]:**

T2 [Հաջորդ նախածանց]:
I←I+1,
J=1,…, K համար կատարել **q[J]←δ[J](q[J], A[I]):**

T3 [Նախածանցի ստուգում]:
J=1, …, K համար կատարել.
եթե **q[J]∈F[J],** ապա
length←I, index←J, անցնել **T4**:

T4 [I=N?]:
Եթե **I=N**, ապա ավարտել աշխատանքը:
Հակառակ դեպքում՝ անցնել **T2**:

14.3. Լեքսիկական վերլուծության խնդրի լուծում

Լեքսիկական վերլուծության խնդրի լուծումը կարող ենք կառուցել՝ պարբերաբար լուծելով առաջին լեքսեմի որոշման խնդիրը հետևյալ կերպ.

Ալգորիթմ H. Լեքսիկական վերլուծություն:

Մուտք:

- Դատարկ բառ չճանաչող վերջավոր ավտոմատների

M[J], 1≤J≤K , հաջորդականություն,

- **W=A[I], 1≤I≤N (N≥0),** բառ Σ այբուբենում:

Ելք:

- **W** բառի լեքսիկական վերլուծության արդյունքում ստացվող զույգերի $(L_1, J_1)...(L_m, J_m)$ հաջորդականություն:

Մեթոդ:

H1 [Սկզբնական արժևորում]:

m←1, h←1:

H2 [Հերթական լեքսեմի որոշում]:

Կիրառել **T** ալգորիթմը վերջավոր ավտոմատների **M[J](1≤J≤K)** հաջորդականության և **W=A[h]...A[N]** բառի նկատմամբ՝ ստանալով **length** և **index** արժեքները:

Կատարել

L_m**←A[h]...A[h+length-1],** J_m**←index;**

H3 [Ավարտի պայման]:

h←h+length:

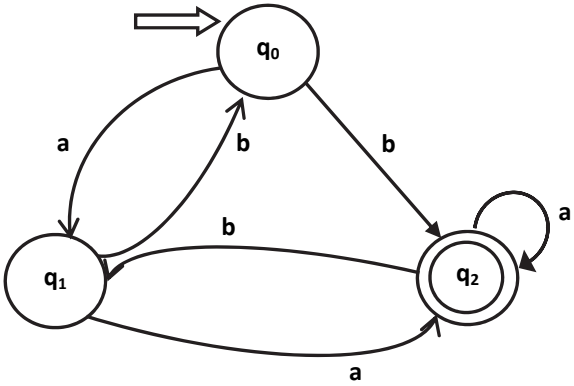
Եթե **h>N**, ապա ավարտել աշխատանքը:

Հակառակ դեպքում՝ կատարել **m←m+1** և անցնել **H2**:

15. Լեքսիկական վերլուծության ծրագիր Prolog լեզվով

15.1. Լեքսիկական վերլուծություն միատիպ լեքսեմների դեպքում

Ստորև կառուցենք լեքսիկական վերլուծության խնդրի լուծման ծրագիր պարզագույն դեպքում, երբ առկա է լեքսեմների ընդամենը մեկ տիպ: Օրինակ, եթե լեքսեմների տիպը ներկայացված է **Նկ. 15.1**-ում բերված ավտոմատի միջոցով, ապա **abbabb** տողի լեքսիկական վերլուծությունը կտա **(abba,1) (b,1) (b,1)** արդյունք:



Նկ. 15.1

Նկատենք, որ **M** ավտոմատով ներկայացված լեքսեմների տիպի դեպքում **w** բառի լեքսիկական վերլուծությունը տալիս է **T** արդյունք այն և միայն այն դեպքում, երբ

- **w**-ն և **T**-ն միաժամանակ դատարկ են,
- **w**-ն **M** ավտոմատով ճանաչվող նախածանց չունեցող ոչ դատարկ բառ է, **T=(w,-1)**,

- **w=tw$_1$, T=(t,1)T$_1$** , որտեղ **t**-ն **M** ավտոմատով ճանաչվող **w** բառի ամենամեծ նախածանցն է, իսկ **T$_1$**-ը՝ **w$_1$** բառի լեքսիկական վերլուծության արդյունքն է:

Դիցուք **lexicalAnalysis(W, Tokens)**-ը պնդում է, ըստ որի՝ դատարկ բառը չճանաչող **M** ավտոմատի միջոցով **W** բառի լեքսիկական վերլուծության արդյունքում ստացվում է զույգերի **Tokens** ցուցակը: Ստորև բերված է **lexicalAnalysis** պրեդիկատի սահմանումը.

```
lexicalAnalysis([], []):- !.
lexicalAnalysis(W, [(T, 1)|Tokens]):-
        firstToken(W, T, W1), lexicalAnalysis (W1, Tokens), !.
lexicalAnalysis(W, [(W, -1)]).
```

Այստեղ **firstToken(W, T, W1)**-ը պնդում է, ըստ որի՝ գոյություն ունեն **M** ավտոմատով ճանաչվող **W** բառի նախածանցներ, և **T**-ն դրանցից ամենամեծն է (առաջին լեքսեմն է), իսկ **W1** բառը ստացվում է **W** բառից **T** նախածանցը հեռացնելու արդյունքում: **firstToken** պրեդիկատը ներկայացնենք հետևյալ կերպ.

```
firstToken(W, T, W1):- startState(Q0), firstToken1(Q0, W, T, W1).
```

որտեղ **firstToken1(Q, W, T, W1)**-ը պնդում է, ըստ որի՝ գոյություն ունեն **M** ավտոմատի **Q** վիճակից ճանաչվող **W** բառի նախածանցներ, և **T**-ն դրանցից ամենամեծն է, իսկ **W1** բառը ստացվում է **W** բառից **T** նախածանցը դեն նետելու արդյունքում: **firstToken1** պրեդիկատը սահմանենք հետևյալ կերպ.

```
firstToken1(Q, [A|W], [A|T], W1):-
                transition(Q, A, Q1), firstToken1(Q1, W, T, W1),!.
firstToken1(Q, W, [], W):- finalStates(F), member(Q, F).
```

15.2. Վերջավոր ավտոմատների զուգահեռ կոմպոզիցիա

15.2.1. Ավտոմատների զուգահեռ կոմպոզիցիա և լեքսիկական վերլուծություն

Լեքսիկական վերլուծությունը բազմատիպ լեքսեմների համար բերենք լեքսիկական վերլուծության միատիպ լեքսեմների համար՝ օգտագործելով ավտոմատների զուգահեռ կոմպոզիցիայի գործողությունը:

Դիցուք տրված է դատարկ բառը չճանաչող վերջավոր ավտոմատների $M_J=(Q_J, \Sigma, \delta_J, q0_J, F_J)$, $1\le J\le K$, հաջորդականությունը: Ըստ ենթադրության՝ $q0_J\notin F_J$, $1\le J\le K$: Կառուցենք $M=(Q, \Sigma, \delta, q0, F)$ վերջավոր ավտոմատը, որտեղ

- $Q=Q_1\times\ldots\times Q_K$,
- $\delta(<q_1, \ldots, q_K>, a)=< q_1', \ldots, q_K'>$, եթե $\delta_1(q_1, a)=q_1', \ldots, \delta_K(q_K, a)= q_K'$, $a\in\Sigma$;
- $q0=<q0_1, \ldots, q0_K>$;
- $F=\{<q_1, \ldots, q_K> \mid \exists j, 1\le j\le K, q_j\in F_j\}$:

M ավտոմատն անվանենք $M_1, \ldots, M_K$ ավտոմատների զուգահեռ կոմպոզիցիա և նշանակենք $M=M_1\times\ldots\times M_K$:

$M_1, \ldots, M_K$ ավտոմատների միջոցով W բառի լեքսիկական վերլուծության արդյունքում ստացվող զույգերի $(L_1, J_1)\ldots (L_m, J_m)$ հաջորդականությունը կարող ենք ստանալ՝ վերլուծելով W բառը $M=M_1\times\ldots\times M_K$ ավտոմատով հետևյալ կերպ.

Քայլ 0: Զույգերի հաջորդականությունը դատարկ է, $s=1$:

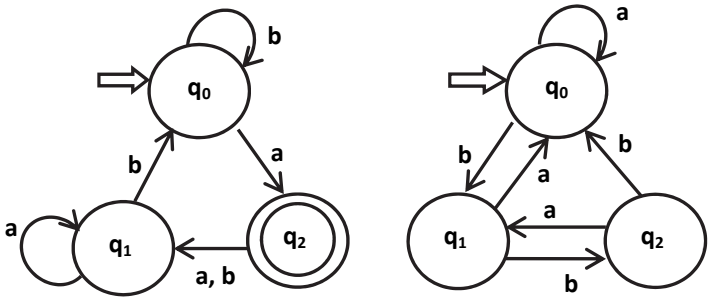
Քայլ s: Դիցուք կառուցվել է զույգերի $(L_1, J_1)\ldots (L_{s-1}, J_{s-1})$ հաջորդականությունը, $W=(L_1\ldots L_{s-1})X$, $s\ge 1$: Կառուցված հաջորդականությանը ավելացնել (L_s, J_s) զույգը, որը որոշվում է հետևյալ կերպ.

եթե գոյություն չունի M ավտոմատով ճանաչվող X բառի նախածանց, ապա վերցնել $L_s=X$, $J_s=-1$: Հակառակ դեպքում,

- նշանակել L_s–ով **M** ավտոմատով ճանաչվող **X** բառի ամենամեծ նախածանցը,
- եթե **M** ավտոմատը, գտնվելով սկզբնական վիճակում և կարդալով L_s բառը, հայտնվում է $<q_1, ..., q_K>$ վերջնական վիճակում, ապա վերցնել $J_s=min\{J| 1\le J\le K, q_J\in F_J\}$.

15.2.2. Ավտոմատների հաջորդականության ներկայացում Prolog լեզվով

Ենթադրենք, որ $M_1, ..., M_K$ ավտոմատների ներկայացման համար օգտագործվում են ընդլայնված **states, symbols, transition, startState** և **finalStates** պրեդիկատները, որոնցում առաջին արգումենտի տեղում նշվում է ավտոմատի կարգահամար.



Նկ. 15.2

Օրինակ՝ **Նկ. 15.2**-ում բերված ավտոմատների հաջորդականությունը կներկայացվի հետևյալ կերպ.

```
/* ***** states ***** */
states(1, [q0, q1, q2]).
states(2, [q0, q1, q2]).

/* ***** symbols ***** */
symbols(1, [a, b]).
symbols(2, [a, b]).
```

```
/* ***** transitions ***** */
transition(1, q0, a, q2).
transition(1, q0, b, q0).
transition(1, q1, a, q1).
transition(1, q1, b, q0).
transition(1, q2, a, q1).
transition(1, q2, b, q1).
transition(2, q0, a, q0).
transition(2, q0, b, q1).
transition(2, q1, a, q0).
transition(2, q1, b, q2).
transition(2, q2, a, q1).
transition(2, q2, b, q0).

/* ***** initial state ***** */
startState(1, q0).
startState(2, q0).

/* ***** final states ***** */
finalStates(1, [q2]).
finalStates(2, [q2]).
```

15.2.3. Ավտոմատների զուգահեռ կոմպոզիցիայի կառուցում

Հիմք ընդունելով վերջավոր ավտոմատների հաջորդականության վերը դիտարկված ներկայացումը՝ կառուցենք դրանց զուգահեռ կոմպոզիցիան **Prolog** լեզվով: Վերջինս ենթադրում է զուգահեռ կոմպոզիցիան ներկայացնող ավտոմատի վիճակների, սիմվոլների, անցումների, սկզբնական վիճակի և վերջնական վիճակների նկարագրման **states, symbols, transitions, startState** և **finalStates** պրեդիկատների սահմանում:

1. **states** պրեդիկատի սահմանում

   states(Q)-ն պնդում է, ըստ որի՝ $M_1 \times \ldots \times M_K$ ավտոմատի վիճակների ցուցակն է **Q**-ն: Ներկայացնենք այս պնդումը հետևյալ կերպ.

   ```
   states(Q):- states1(1, Q).
   ```

   որտեղ **states1(J, Q)**-ն պնդում է, ըստ որի՝ **Q**-ն $M_J \times \ldots \times M_K$ ավտոմատի վիճակների ցուցակն է: **states1** պրեդիկատը սահմանենք հետևյալ կերպ.

   ```
   states1(J, Q):- states(J, S),
                   J1 is J+1, states1(J1, Q1),combine(S, Q1, Q),!.
   states1(J, Q):- states(J, S), convert(S, Q).
   ```

   Այստեղ օգտագործված **combine** և **convert** պրեդիկատներն ունեն հետևյալ իմաստը:

   - Վիճակների **S** և վիճակների վեկտորների **Q1** ցուցակների համար **combine(S, Q1, Q)**-ն պնդում է, ըստ որի՝ վիճակների վեկտորների **Q** ցուցակը ստացվում է **Q1**-ին պատկանող բոլոր վեկտորների առջևում **S**-ին պատկանող բոլոր տարրերի կցագրելու արդյունքում: **combine** պրեդիկատի սահմանումն է.

     ```
     combine([], _, []):- !.
     combine([X|S], Q1, Q):- combine1(X, Q1, Q2),
                             combine(S, Q1, Q3), append(Q2, Q3, Q), !.
     combine1(_, [], []):- !.
     combine1(X, [V|Q1], [[X|V]|Q]):- combine1(X, Q1, Q).
     ```

   - Վիճակների **S** ցուցակի համար **convert(S, Q)**-ն պնդումն է, ըստ որի **Q** ցուցակը ստացվում է **S** ցուցակից վերջինիս տարրերը միատար ցուցակների վերածելու արդյունքում: **convert** պրեդիկատի սահմանումն է.

     ```
     convert([], []):-!.
     convert([X|S], [[X]|Q]):- convert(S, Q).
     ```

2. **symbols** պրեդիկատի սահմանում

   Հաշվի առնելով, որ զուգահեռ կոմպոզիցիա հանդիսացող ավտոմատի և դրա բաղադրիչ ավտոմատների մուտքային սիմվոլների ցուցակները համընկնում են, ստանում ենք.

   symbols(Sigma):- symbols(1, Sigma).

3. **transitions** պրեդիկատի սահմանում

   transitions(Delta)-ն պնդում է, ըստ որի՝ **Delta**-ն $\mathbf{M_1 \times \ldots \times M_K}$ ավտոմատի անցումների ցուցակն է: Ներկայացնենք այս պնդումը հետևյալ կերպ.

   transitions(Delta):- states(Q), symbols(Sigma),
   subTransitions(Q, Sigma, Delta).

   որտեղ **subTransitions(Q, Sigma, Delta)**-ն պնդում է, ըստ որի՝ անցումների **Delta** ցուցակը կազմված է $\mathbf{M_1 \times \ldots \times M_K}$ ավտոմատի այն անցումներից, որոնցում սկզբնական վիճակը պատկանում է **Q**-ին, իսկ ընթերցվող սիմվոլը՝ **Sigma**-ին: **subTransitions** պրեդիկատը սահմանենք հետևյալ կերպ.

   subTransitions([], _, []):- !.
   subTransitions([V|Q], Sigma, Delta):-
   transitionsFromState(V, Sigma, Delta1),
   subTransitions(Q, Sigma, Delta2),
   append(Delta1, Delta2, Delta), !.

   Այստեղ **transitionsFromState(V, Sigma, Delta)**-ն պնդում է, ըստ որի՝ **Delta**-ն կազմված է է $\mathbf{M_1 \times \ldots \times M_K}$ ավտոմատի այն անցումներից, որոնցում սկզբնական վիճակն է **V**-ն, իսկ ընթերցվող սիմվոլը պատկանում է **Sigma**-ին: **transitionsFromState** պրեդիկատի սահմանումն է.

   transitionsFromState(_, [], []):-!.
   transitionsFromState(V, [A|Sigma], [(V, A, V1)|Delta]):-
   transition(V, A, V1), transitionsFromState(V, Sigma, Delta).

Վերջապես **transition(V, A, V1)**-ը պնդում է, ըստ որի՝ $M_1 \times \ldots \times M_K$ ավտոմատը, **V** վիճակից կարդալով **A** սիմվոլը, անցնում է **V1** վիճակին: Այս պրեդիկատի սահմանումն է.

```
transition(V, A, V1):- transition1(1, V, A, V1).
transition1(_, [], _, []):- !.
transition1(J, [X|V], A, [Y|V1]):-
                                transition(J, X, A, Y),
                                J1 is J+1,
                                transition1(J1, V, A, V1).
```

որտեղ **transition1(J, V, A, V1)**-ը պնդում է, ըստ որի՝ $M_J \times \ldots \times M_K$ ավտոմատը **V** վիճակից կարդալով **A** սիմվոլը անցնում է **V1** վիճակին:

4. **startState** պրեդիկատի սահմանում
   startState(V)-ն պնդում է, ըստ որի՝ **V**-ն $M_1 \times \ldots \times M_K$ ավտոմատի սկզբնական վիճակն է: Ներկայացնենք այն հետևյալ կերպ.

   ```
   startState(V):-startState1(1, V).
   ```

   որտեղ **startState1(J, V)**-ն պնդում է, ըստ որի՝ **V** –ն $M_J \times \ldots \times M_K$ ավտոմատի սկզբնական վիճակն է: Սահմանենք **startState1** պրեդիկատը հետևյալ կերպ.

   ```
   startState1(J, [X|V]):-    startState(J, X),
                              J1 is J+1,
                              startState1(J1, V), !.
   startState1(J, [X]):-      startState(J, X).
   ```

5. **finalStates** պրեդիկատի սահմանում
   finalStates(F)-ը պնդում է, ըստ որի՝ **F**-ը $M_1 \times \ldots \times M_K$ ավտոմատի վերջնական վիճակների ցուցակն է: Ներկայացնենք այն հետևյալ կերպ.

   ```
   finalStates(F):- states(Q), subFinalStates(Q, F).
   ```

որտեղ **subFinalStates(Q, F)**-ը պնդում է, ըստ որի՝ **F** ցուցակի մեջ ընդգրկված են **Q** ցուցակին պատկանող $M_1 \times \ldots \times M_K$ ավտոմատի վերջնական վիճակները: **subFinalStates** պրեդիկատը սահմանենք հետևյալ կերպ.

```
subFinalStates([], []):- !.
subFinalStates([V|Q], [V|F]):-
                        firstFinal(V, _), subFinalStates(Q, F),!.
subFinalStates([_|Q], F):- subFinalStates(Q, F).
```

Այստեղ **firstFinal(V, L)**-ը պնդում է, ըստ որի՝ **V** -ն $M_1 \times \ldots \times M_K$ ավտոմատի վերջնական վիճակ է, $L=\min\{I \mid 1 \le I \le K,\ V_L \in F_L\}$. Ներկայացնենք այն հետևյալ կերպ.

```
firstFinal(V, L):-      firstFinal1(1, V, L).
firstFinal1(J, [X|_], J):-  finalStates(J, F), member(X, F), !.
firstFinal1(J, [_|V], L):-  J1 is J+1,
firstFinal1(J1, V, L).
```

որտեղ **firstFinal1(J, V, L)**-ը պնդում է, ըստ որի՝ **V** -ն $M_J \times \ldots \times M_K$ ավտոմատի վերջնական վիճակ է, $L=\min\{ I \mid J \le I \le K,\ V_L \in F_L\}$.

15.3. Լեքսիկական վերլուծություն ընդհանուր դեպքում

Ավտոմատների զուգահեռ կոմպոզիցիայի օգտագործումը հնարավորություն է տալիս լեքսիկական վերլուծության խնդիրը բազմատիպ լեքսեմների դեպքում բերելու լեքսիկական վերլուծության խնդրին միատիպ լեքսեմների դեպքում, եթե հերթական լեքսեմը որոշելու հետ մեկտեղ որոշվի նաև դրա տիպը: Վերջինիս համար բավարար է վերլուծել զուգահեռ կոմպոզիցիան ներկայացնող ավտոմատի վերջնական վիճակը՝ բացահայտելու համար վերջնական վիճակ հանդիսացող առաջին բաղադրիչը:

Ստորև բերվում է բազմատիպ լեքսեմների դեպքում լեքսիկական վերլուծություն կատարող ծրագիրը:

```
lexicalAnalysis([], []):- !.
lexicalAnalysis(W, [(T, J)|Tokens]):- firstToken(W, T, J, W1),
                                      lexicalAnalysis(W1, Tokens), !.
lexicalAnalysis(W, [(W, -1)]).

firstToken(W, T, J, W1):-  startState(V),
                           firstToken1(V, W, T, J, W1).

firstToken1(V, [], [], J, []):-  firstFinal(V, J), !.
firstToken1(V, [A|W], [A|T], J, W1):-transition(V, A, V1),
                                      firstToken1(V1, W, T, J, W1),!.
firstToken1(V, W, [], J, W):-  firstFinal(V, J).
```

16. Շարահյուսական վերլուծություն Prolog լեզվով

Շարահյուսական վերլուծության ժամանակ անհրաժեշտ է պարզել՝ համապատասխանում է արդյոք լեքսեմների տրված հաջորդականությունը լեզվի քերականությանը, թե ոչ: Հաշվի առնելով, որ ծրագրավորման լեզուների շարահյուսությունը որպես կանոն նկարագրվում է կոնտեքստից ազատ քերականությամբ (*Բեկուսի-Նաուրի* նոտացիա), որտեղ քերականության հիմնական սիմվոլների դերում հանդես են գալիս լեքսեմները, իսկ օժանդակ սիմվոլների դերում՝ շարահյուսական միավորները (*արտահայտություն*, *հրահանգ*, *ծրագիր* և այլն), շարահյուսական վերլուծության խնդիրը կարելի է սահմանել հետևյալ կերպ.

Տրված են.

1. **G=(N, Σ, P, S)** առանց ձախ անդրադարձման կոնտեքստից ազատ քերականությունը,
2. **W** բառը **Σ** այբուբենում:

   Պահանջվում է որոշել՝ պատկանում է արդյոք **W** բառը **G** քերականությամբ ծնվող լեզվին, թե ոչ:

Այսպիսով շարահյուսական վերլուծության խնդիրը բերվում է **13.2** բաժնում դիտարկված կոնտեքստից ազատ քերականությամբ ծնվող լեզվին բառի պատկանելիության խնդրին: Արտաբերենք այնտեղ ստացված լուծումը՝ փոքր ինչ ձևափոխված տեսքով.

```
parsing(W):-  initialSymbol(S), generates([S], W).


generates([], []):-!.
generates([X|Alpha],[X|W]):-  !, generates(Alpha, W).
generates([A|Alpha], W):- production(A, Right),
        append(Right, Alpha, Alpha1), generates(Alpha1, W),!.
```

Հավելված 1

Նախագծերի թեմաներ

1. *Գրաֆների հետ կապված խնդիրներ*
   1.1. Մինիմալ կմախքային ծառի կառուցում Պրիմի ալգորիթմով:
   1.2. Մինիմալ կմախքային ծառի կառուցում Կրասկալի ալգորիթմով:
   1.3. Կարճագույն ճանապարհների կառուցում Դեյկստրայի ալգորիթմով:
2. *Վերջավոր ավտոմատներ և կանոնավոր լեզուներ*
   2.1. Չդետերմինացված վերջավոր ավտոմատին համարժեք կանոնավոր արտահայտության կառուցում:
   2.2. Չդետերմինացված վերջավոր ավտոմատին համարժեք դետերմինացված վերջավոր ավտոմատի կառուցում:
   2.3. Վերջավոր դետերմինացված ավտոմատի մինիմիզացում:
3. *Կոնտեքստից ազատ քերականություններ*
   3.1. Կոնտեքստից ազատ քերականությունից անօգտակար սիմվոլների, **ε**- և շղթայական կանոնների հեռացում:
   3.2. Կոնտեքստից ազատ քերականության բերում Խոմսկու նորմալ ձևին:
   3.3. Կոնտեքստից ազատ քերականության բերում Գրեյբախի նորմալ ձևին:
4. *Այլ խնդիրներ*
   4.1. Մեծ թվերի թվաբանության մոդելավորում (գումարում, հանում, բազմապատկում, ամբողջաթիվ բաժանում, բաժանման մնացորդի հաշվում): **100!**-ի հաշվում:
   4.2. Արտահայտությունների ածանցում:
   4.3. «Կետ-խաչ» խաղի ծրագրավորում:

ԳՐԱԿԱՆՈՒԹՅՈՒՆ

1. **W. F. Clocksin, C. S. Mellish.** Programming in Prolog, 5th edition, Springer, 2003 /Русский перевод: *У. Клоксин, К. Мелиш. Программирование на языке Пролог, 2005*/.
2. **I. Bratko.** Prolog Programming for Artificial Intelligence, 4th edition, Pearson Education Canada, 2011 /Русский перевод: *И. Братко. Алгоритмы искусственного интеллекта на языке Пролог. 3-е издание, Издательский дом "Вильямс", 2004*/.
3. **L. Sterling, Eh. Shapiro.** The Art of Prolog, 2nd edition, MIT Press, 1999 /Русский перевод: *Л. Стерлинг Л., Э. Шапиро. Искусство программирования на языке Пролог*/.
4. **M. Bramer.** Logic Programming with Prolog, 2nd edition, Springer, 2013.
5. **L. Homem.** Topics in Programming Languages, Chartridge Books Oxford, 2013.

ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ
ՏԵՂԵԿԱՏՎԱԿԱՆ ՏԵԽՆՈԼՈԳԻԱՆԵՐԻ ԿՐԹԱԿԱՆ ԵՎ
ՀԵՏԱԶՈՏԱԿԱՆ ԿԵՆՏՐՈՆ

ԱՐՄԵՆ ՀՐԱՉԻ ԿՈՍՏԱՆՅԱՆ

ԽՆԴԻՐՆԵՐԻ ԼՈՒԾՈՒՄ **PROLOG** ԼԵԶՎՈՎ

Ուսումնամեթոդական ձեռնարկ

Համակարգչային ձևավորումը՝ Կ. Չալաբյանի
Կազմի ձևավորումը՝ Ա. Պատվականյանի
Հրատ. սրբագրումը՝ Մ. Հովհաննիսյանի

Տպագրված է «Գևորգ-Հրայր» ՍՊԸ-ում:
ք. Երևան, Գրիգոր Լուսավորչի 6

Ստորագրված է տպագրության 15.09.2016:
Չափսը՝ 60x84 1/16: Տպ. մամուլը՝ 7,125:
Տպաքանակը՝ 100 օրինակ:

ԵՊՀ հրատարակչություն,
ք. Երևան, 0025, Ալեք Մանուկյան 1

ՀՐԱՏԱՐԱԿՉՈՒԹՅՈՒՆ
ԵՐԵՎԱՆ 2016
publishing.ysu.am