КОНСПЕКТ

I. Теоритични основи-семантика на езици за програмиране(ЕП).

1. Основни понятия
2. Виртуална изчислителна машина(ВИМ)
3. Език
4. Мета-Език на Бекус-Наур
5. Транслация(методи и схеми)
6. Свързване и време на свързване
7. Базова ВИМ
8. Базов ЕП()
9. Данни във входния език за програмиране
10. Операции във входния език за програмиране
11. Управление на последователността от действия(УПД)
12. Управление на данните
13. Управление на паметта
14. Операционна среда

II. Техники на реализация на транслатори

15. Сентактични елементи на език за програмиране
16. Синтактичен анализ на програма(разпознаване)
17. Етапи на транслация
18. Лексикален анализ
19. Синтактичен анализ
20. Семантичен анализ
21. Интерпреация на програма
22. Генерация на код
23. Оптимизация
24. Схеми за реализация на транслатор

1. Основни понятия

Език за Програмиране – Наричаме всяка система от означения, предназначена за описание на алгоритми и структури от данни, която е реализарана на изчислителна машина. Ако този език не е реализиран(не се използва) се нарича алгоритмичен език. Реализацията на даден език се заключава в създаването на транслатор, който превежда програмите, написани на този език в еквивалентни програми, написани в термините на езика на изчислителната машина, т.е. в програми, които могат да се изпълняват непосредствено от изчислителната машина.

Програма – Наричаме алгоритъм и използваните от него стуктури от данни, описани в термините на определен език за програмиране.

Изчислителна Машина – Наричаме интегрирано множество от алгоритми и структури от данни, способни да съхраняват и изпълняват програми. Всяка изчислителна машина може да бъде построена по два начина:
А) Като реално физическо устройство.
Пр. Компютър без Операционна система.
Б) Като програма, изпълнявана от друга изчислителна машина.

Ние не обсъждаме начина на реализация на изчислителната машина. Поради това ще използваме термина виртуална изчислителна машина.

Транслация – Наричаме процесът на превод на програма, написана в термините на езика на една виртуална изчислителна машина в еквивалентна програма, написана в термините на друга тъкава.

От тези дефиниции следва, че изчислителния език е и ВИМ. Целта на този курс лекции е системното обсъждане на дейността транслация, т.е. :
А)Кой е обекта на дейността?
Б)Какъв е резултатът от дейността?
В)Кой извършва дийността?
Г)Как я извършва?
Д)Защо я извършва?
Е)как да направим модел на извършителя на дейността на транслтор?

Курсът лекции се базира върху трите работни понятия:
1.Алгоритъм.
2.Структура от данни.
3.Език.
Както и върху методологичното понятие система(системен подход).

Обсъжданията ще бъдат предопределени от една-единствена цел - създаване на виртуална изчислителна машина, реализираща дейността транслация. Това налага понятието Виртуална Изчислителна Машина да бъде обсъдено в шест теоритично-обусловени и взаимно свързани аспекта:
1.Данни.
2.(Елементарни) Операции.
3.Управление на последователността на действия.
4.Управление на данните.
5.Управление на паметта.
6.Операционна среда.

Понятието програма може и трябва да разглеждаме като симоним на понятието Абстрактна Виртуална Изчислителна Машина. Тъй като всеки, който владее даден език за програмиране(познава семантиката на езика за програмиране), може да изпълни произволна правилан програма, написана на този език.

Изводът е, че всеки език за програмиране трябва да бъде обсъден в термините на така въведените шест аспекта.

2.Виртуална изчислителна машина(ВИМ)

I. Данни

Всяка изчислителна машина, която да предоставя различни типове данни(прости и съставни), в смисъл, че може да изпълнява определено количество операции върху тях. Обикновено ние свързваме понятието данни с начини на съхраняване в оперативната памет, бързите регистри или външната памет на компютрите. Това не е правилно, тъй като компютърът е просто една конкретна физическа реализация на дадена абстрактна виртуална изчислителна машина(АВИМ). Друг е въпросът, че трябва да се съобразяваме с този аспект на типовете данни, тъй като преведените програми се изпълняват именно на компютър. Изводът е – понятието данни(данна) трябва да се разглежда като конкретно представяне на даден тип данни. Следователно различните виртуални изчислителни машини(ВИМ) могат да предлагат един и същ тип данни, но да обработват различни представяния на този тип данни. В този случай казваме, че компютрите имат различни вградени типове данни. Понетието данни не трябва да се разглежда само в аспект – обекти, върху които се прилагат операции. Това означава, че данните мога да описват операции, следователно всяка виртуална изчислителна машина трябва да предоставя и особен тип данни, а именно данни за описание на операциите, които тя може да изпълнява. Тези данни обикновено наричаме **команди**. Ако това е налице,т.е. има такъв тип данни, ВИМ се нарича универсална, тъй като може да изпълнява различни програми. В противен случай тя се нарича **специализирана**.

Каква ВИМ е транслатора?

Задача: Дефинирайте понятието тип данни.

II. Операции

Всяка ИМ трябва да предоставя множество от елементарни операции за манипулиране и обработка на вградените й типове данни. Съществено е, че понятието елементарна операция е относително, т.е. дадена елементарна операция сама по себе си може да се реализира от произволно сложна ВИМ.

а) Дадена операция може да бъде класифицирана като елементарна само по отношение на дадена ВИМ.

б) Две елементарни операции в две различни ВИМ са еквивалентни тогава и само тогава, когато реализиращите ги ВИМ са тъждествени(като алгоритми);

в) Кога две елементарни операции са тъждествени?

III. Управление на последователността от действия (УПД)

Всяка универсална ВИМ трябва да предоставя средства за управление на последователността на изпълнение на елементарните й операции. В противен случай ВИМ ще изпълнява само линейни алгоритми. Какъв механизъм за УПД е възприет в компютрите на ниво машинен език(асемблер). В ЕП от високо ниво се използват три механизма за УПД.

а) За УП от операции в (аритметични) изрази;

б) За УП на изпълнение на оператори(команди);

в) За УП на изпълнение на подпрограми;

Що е подпрограма? – част от програмата със същата структура, зависеща от параметри.

IV. Управление на Данни (УД)

Най-общо под УД във ВИМ разбираме механизъм за достъп до данните. При наличието на такъв механизъм, операциите : четене на данни(read) и операцията замяна да данна(replace) са достатъчни за реализиране на произволен алгоритъм за работа с данни. Класическият механизъм за УД свързва с всяка данна уникално име, чрез което данната се разпознава, т.е. достъпът до данна се осъществява посредством името й. Този класически механизъм има множество модификации, опередeлени от момента на свързване, типа на връзката и продължителността на връзката между данната и името й. В най-простия нариант на този механизъм:

а) Под име на данна се разбира адрес, т.е. място в средата за съхраняване на данни;

б) Типът на данната с определено име се определя от операцията, която я използва;

в) Съответствието данна-име е взаимно еднозначно;

По какво се различава механизма за УД в език от високо ниво от този прост механизъм?

V. Управление на паметта (УП)

Всяка ВИМ разполага със среда за съхранение на програми и данни. Този ресурс е краен, поради което се налага той да бъде преразпределен по време на изпълнение на дадена програма. В класическите ВИМ такъв механизъм липсва, т.е. програмата и данните й се намират на фиксирани места през цялото време на изпълнение на програмата.

VI. Операционна среда (ОП)

Всяка ВИМ функционира в някаква външна(операционна) среда, т.е. операционната среда са външни за ВИМ обетки, с които тя взаимодейства, като взаимодействието се свежда до получаване и изпращане на данни. Тези външни за ВМ обекти трябва да се разглеждат също като ВИМ, т.е. всяка ВИМ комуникира с други ВИМ посредством обмен на данни. Операционна среда на ВИМ наричамесъвкупността от ВИМи, с които тя комуникира.

Що е Операционна Система?

VII. Динамика на ВИМ

Разгледаните шест аспекта ни дават статичната организация на ВИМ. За да разберем динамиката на ВИМ въвеждаме понятието **състояние на ВИМ.** Като го определяме като състояние на съхраняващата среда на ВИМ. Следователно всяка изпълнена от ВИМ операция

променя състоянието на ВИМ. По такъв начин изпълнението на една програма може и трябва да се разглежда като последователност от състояния на ВИМ.

Казваме, че разбираме една програма ако можем да предскажем последователността от състояния на ВИМ, която би се получила ако програмата се изпълни с тези данни.

Как търсим грешка в програмата?

Що е грешка в програмата?

3. Език

Езикът е система от съобщения, съставени от някакви знаци, чрез които може да се обменя информация.
Езикът е система от изразни средства и правила за ползването им, чрез която може да се представя и обменя информация под формата на дискретни съобщения.
Според начина на формирането си езиците се делят на **естествени** и **изкуствени**, а според формата на съобщенията на **писмени** и **говорими**.
Като система всеки език има определена структура, най-често с йерархичен характер. Най-ниското ниво е азбуката на езика и това е множество от неделими и атомарни елементи на езика. На следващото ниво са думите(словосъчетанията). Те образуват речниковия(лексикалния) фонд на езика. При естествените езици няма строги правила, по които се формират лексемите(думите). Следващото ниво са изреченията. Те са структури от лексеми, изградени съгласно някакви правила. Съвкупността от тези правила се нарича синтаксис на езика. Правилата, по които знанията за реалността се представят(преобразуват) като съобщения на езика и обратно се нарича **семантика на езика**. Няколко езика образуват група езици ако имат сходна лексика, синтаксис и семантика. **Мета език(над език)** наричаме всеки език, които служи за описание на друг език. Естествените езици са мета езици.
Формална граматика наричаме мета език за описание на синтаксиса на даден език L.
Формалната граматика G се дефинира като наредената четворка (A, T, P, Z), където:

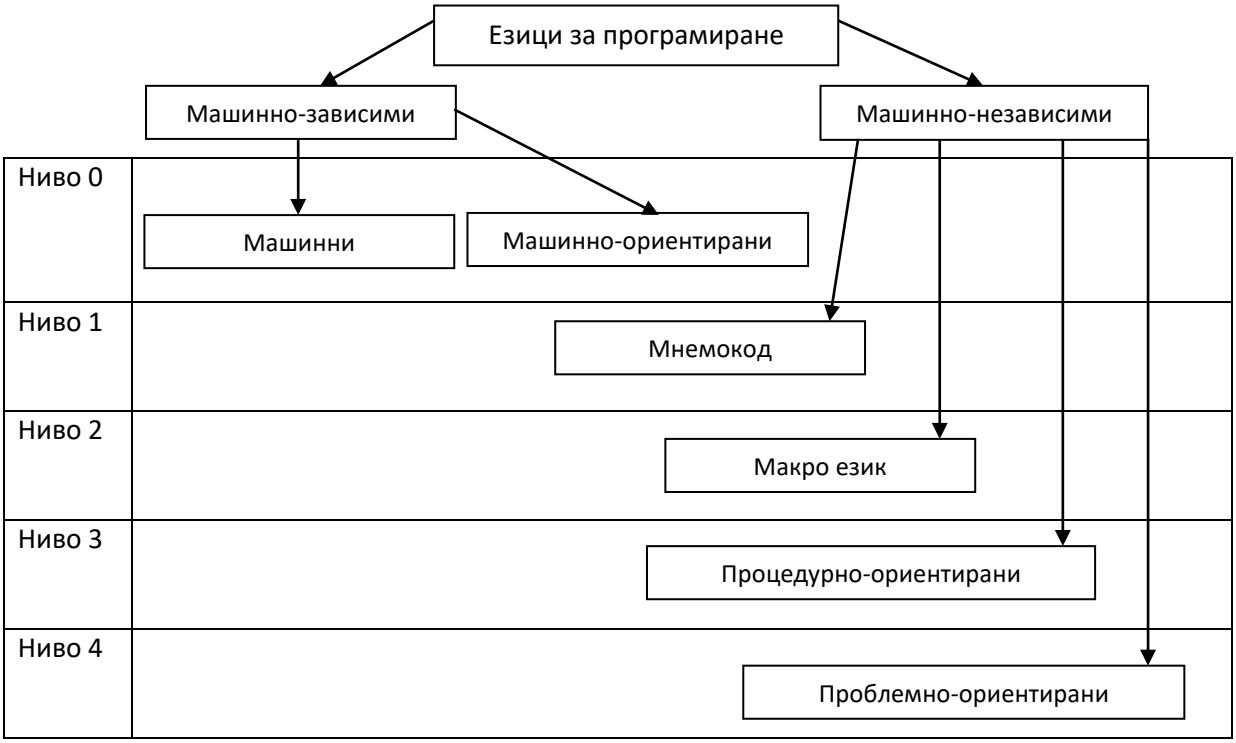
A – азбука на G;
T подмножество на A;
P е крайно множество от правила на езика, в смисъл – правила за образуване на думи и изречения;
Z принадлежи на A\T и се нарича аксиома;

Формален език L, породен от G, се нарича множеството от всички низове, изградени от терминални знаци, които могат да се изведат от аксиомата Z. Обсъжданите от нас езици за програмиране са породени от така наречените регулярни граматики. Това са граматики, чиито правила за извод имат вида:

U :: =N и U :: = W N , където U, W принадлежи на A\T и W принадлежи на T. – Чомски(Хомски)

В езиците за програмиране се различават следните съставни елементи(подсистеми):
а) Азбука, схормирана от различни подмножества знаци???
б) Лексеми, които биват два типа – идентификатори(имена в собствен смисъл) и литерали;
в) Прости изречения, които са изрази или прости оператори;
г) Съставни изречения, които биват съставни оператори, подпрограми, класове, модули и т.н.;
д) Програми – низ, изводим от аксиомата на граматика;

Според нивото си на близост до естествния език, езиците за програмиране се делят на пет групи, илюстрирани от следната схема:



4. Мета-Език на Бекус-Наур

Ще опишем неформално граматиката G, която поражда нормалната форма на Бекус-Наур, която описва синтаксиса на език за програмиране. Азбуката на тази граматика включва три групи знаци:

а) **Собствени знаци** –

::= („по дефиниция е”);
| („или”);
<, > ;

б) **Знаци за именуване на променливи** – Всички знаци, чрез които може да се изрази мисъл на български език;

в) **Терминални знаци** – Знаците на езика L, чийто синтаксис ще се дефинира;

Очевидно е, че терминалните знаци не трябва да имат сечение със собствените знаци на езика.
Лексемите на езика, породен от тази граматика G, са два вида:
а) **Литерал** – низ от терминални знаци. Стойността на всеки литерал е самият низ;
б) **Понятие** – това е низ от знаци за именуване на понятия, заграден в Word скоби;
Пр. <програма>, <константа>, <3.14>

Стойността на всяко понятие е множество от литерали. Понятието <ПРАЗНО> обузначава празно множество. Изразите на езика, породен от тази граматика са два вида:
а) **Прост израз**, който е низ от лексеми. Стойността на всеки прост израз е множество от низове от терминални знаци, получено като резултат от Декартовото Произведение на множествата от стойности на всички лексеми, формиращи простия израз. Обикновено стойността на простия израз се изчислява отляво надясно;
б) **Израз** – прост израз или последователност от прости изрази, разделени с знака |(или). Стойността на всеки израз е множеството от низове от терминални знаци, получени в резултат на обединението на множествата от стойности на простите изрази, формиращи израза. Всеки прост израз в израз се нарича алтернатива в този израз;
Оператора на езика, породен от тази граматика е един и се нарича **металингвистична формула**. Той има вида:

<понятие> ::= <израз>

Смисълът на всяка **металингвистична формула**, че всяко понятие се свързва (асоциира, присвоява) множеството от стойности, дефинирани от израза <понятие> ::= <израз>. Казано по друг начин, с метода на металингвистичната формула се дефинира понятие.
Присвоената(асоцииранатa стойност) става текуща стойност на понятието.
Можем ли да кажем, че понятието металингвистична формула е променлива величина от тип многжество в оператор за присвояване. Следователно при всяка металингвистична формула първо се изчислява стойността на израза, на базата на текущите стойности на понятията, които го формират и след това тази стойност се асоциира с понятието. Това означава, че могат да се използват рекурсивни металингвистични формули. Това са формули, чиито понятия участват в формирането на израза. Ако едно понятие се формира чрез пряка или коствена рекурсия, то израза на тази металингвистична формула трябва да съдържа поне една напълно определена алтернатива. Една алтернатива е напълно определена ако при изчисляването на нейната стойност не се използват понятието на металингвистична формула. Често, за да се избегне описването на рекурсията в даден вид се използват знаци за съкратено описание, например:

<дума> ::= <буква> | <дума><буква>, се записва съкратено като:

$$\text{<дума> ::= <буква> } [\text{<буква>}]_{1}^{\infty}$$

Чрез описаната граматика може и се поражда език, съдържащ безкрайно много металингвистични формули. Съществено е, че може да формираме краино множество от металингвистични формули, което да дефинира граматика G, пораждаща езика за програмиране L. Например:

<знак> ::= +| -
<цифра> ::= 0|1
<ЦДЧ> ::= <цифра> | <знак> <цифра> | <ЦДЧ> <цифра>

Това множество, дефинирано формира граматиката G, където:

A={<знак>, <цифра>, <ЦДЧ>, 0, 1, +, -};
T={0, 1, +, -};
P= <ЦДЧ>←----------<цифра>
<ЦДЧ>←----------<знак><цифра>
<ЦДЧ>←----------<ЦДЧ><цифра>
Z=<ЦДЧ>;

5.Транслация(методи и схеми)

Транслация наричаме процеса на превод на програма, написана в термините на езика на една ВИМ и този език обикновенно наричаме входен език в еквивалентна програма написана в термините на езика на друга ВИМ и този език наричаме изходен език(обектен език). Крайната цел на транслирането е преведената програма да се изпълни. Следователно изпълнението на тази преведена програма :

a) Трабва да се разглежда като възможен етап от дейността транслация.
b) Би могло да се разглежда като своеобразен превод от езика на входните данни в термините на езика на изходните данни, т.е преведената програма се разглежда като своеобразен преводач или транслатор.

Има две основни схеми за транслация съобразно това дали транслаторът е и изпълнител или не :

a) Транслация(Компилация) – Ако преводачът не изпълнва преведената програма, процесът на превод се нарича транслация, а преводачът транслатор. В съответствие с нивата на входният и изходният езици се различават :

- **Конвертор** – транслатор на който входният и изходният езици са ЕП от високо ниво.
- **Компилатор (compile)** – това е транслатор на който входният език е ЕП от високо ниво, а изходният език е машинно ориентиран. Терминът компилатор отразява факта, че входната програма може да се състои от различни части, които компилаторът стартира в една изходна програма.
- **Макро Асемблер** – транслатор чиито входен език е макро език, а изходният е модификация на машинния език в който командите ползват само относителна адресация, т.е преместваем формат на обектната програма). Освен това обектната програма съдържа таблици, описващи онези програмни обекти и връзки между тях чието конкретизиране би превърнало изходната програма в изпълнима програма, т.е в програма на машинен език. Входната програма също може да се състои от части(примерно стандартна библиотека).
- **Асемблер** – транслатор, който се отлечава от макро асемблера само по това, че входният му език е мнемо код.
- **Свързващ редактор(редактор на връзките или limker)** –това е транслатор, чиито входен език съвпада с изходния език на асемблера, а изходният му език е машинен език в преместваем формат, т.е limkera обработва таблиците.
- **Зареждач (loader)** - транслатор, чиито входен език съвпада с изходния език на свързващия редактор, а изходният му език е машинен език, т.е изходната прогрма може да си използва. Изходната прогрма на зареждача е записана в оперативната памет.

- **Изпълнител (Executor)** – Това е самата ВИМ.

Предмет на нашия курс лекции са компилаторите и техниките за тяхното реализиране.

b) Интерпретация(Програмно Моделиране) – Програма, написана на машинен език, т.е ВИМ която изпълнява програма написана на друг език, различен от машинния, се нарича интерпретатор , а работата й интерпретация на входният език.

Може ли интерпретаторът да бъде написан на не машинен език?

Според нивото на входният език се различават два типа интерпретатори :

a) Интерпретатор в собствен смисъл когато входният език е език от високо ниво
b) Емулатор - когато входният език е машинно ориентиран

Сравняваме Компилацията и Интерпретацията

Компилация	**Интерпретация**
1. Превод 2. Всяка входна програма се обработва в съответствие с така наречното правило за четене 3. Всяка инструкция или команда на входната програма се обработва фиксиран брой пъти, определен от възприетият метод на транслация(различаваме еднопасови, двупасови многопасови) 4. Входната програма се превежда веднъж 5. Преведената или изходната програма заема предварително не малко място 6. Преведената програма се изпълнява мвного по-бързо в сравнение с евентуалното интерпретиране	1. Превод и изпълнение 2. Входната програма се обработва в съответствие с така наречената логика на изпълнение 3. Всяка инструкция на входната програма се обработва толкова пъти колкото налагат входните данни 4. Входната програма се превежда всеки път когато трябва да се изпълни 5. Не е необходимо място за преведената програма, а евентуално само за данните които тя обработва 6. Изпълнението на входната програма е много по-бавно в сравнение с изпълнението на евентуално компилираната входна програма

Защо интерпретирането е винаги по-бавно в сравнение с изпълнението на компилиранат входна програма?

Очевидно е ,че двете схими на работа имат почти противоположни достойнства(по отношение на зяета ОП и скорост на изпълнение). Поради това особено сега тези две “чисти” схеми се използват много рядко в този си вид. Обикновено се използва така наречената комбинирана схема, т.е входната програма се компилира до подходящ

машинно ориентиран език, а след това получената програма се интерпретира,т.е емулира. Пример в това отношение е езикът Java и така нареченият байт код.

Колко езика трабвя да владее разработчикът на транслатори?

Ако реализационният език на един трансратор е различен от изходният му език, то транслаторът му се нарича **крос-транслатор**. Казано по друг начин, изходната програма трябва да се изпълнява от друга ВИМ, различна от тази на която се изпълнява транслатора.
Трябва да се определи, че входният език(с много редки изключения) не предопределя изборът на метод или схема за транслация, т.е дали езикът ще се реализира чрез транслатор или чрез интерпретатор. Това позволява за даден входен език :

1. Да се разработи интерпретатор реализиращ входният език в термините на реализационния език на интерпретатора и в последствие този интерпретатор да се доразвие до компилатор превеждащ от реализационния език в термините на изходният език. Тази схема на работа ще бъде използвана в упражненията и е предмет на втората част на лекциите.

6.Свързване и време на свързване

Понятуята свързване и време на свързване съпътстват всяка поддейност на дейността програмиране. Тук под поддейност разбираме дейности като : създаване на език, реализация на език, тестване на програма, изпълнение на програма и т.е. Свързвне наричаме дейността от преписване на някаква характеристика на някакъв програмен обект. Характеристиката се избира от някакво предварително известно множество от характеристики(своиство). Казва се още, че обекта се асоциира с характеристиката или по-точно характеристиката се асоциира с обекта. Самата връка се нарича асоцияция. Под програмен обект разбираме име или идентификатор, а всичко друго, което свързваме или асоцииираме с това име, като описание на операцията свързване. Време на свързване наричаме момента от времето, в който става това преписване на характеристика. Опитите за класификация на операцията свързване ще ни отведат извън сферата на дейността програмиране. Поради това ще обсъдим няколко основни класа времена на свързване, имащи пряко отношение към дейността програмиране така, както е дефинирана.

1. **Свързване при дефиниране на езика –** По същество всеки разработчик на език за програмиране фиксира състава и структурата на език, като определя различни типове множества от програмни елементи или допостимите връзки или отношения между тях. По същество това означава, че с обект, наричан език за програмиране се свързват някакви характеристики. Очевидно е, че програмиста, който използва този език не може да излезе извън рамките на възможностите на езика, наложени от неговия разработчик.
2. **Свързване при реализация на езика за програмиране -** Във всеки език за програмиране има елементи, които не са конкретизирани по време на неговото дефиниране. Най-очевидният пример са представянията на числата и операциите върху

тях. С тези елементи се свързват определени характеристики по време на резлизацията на езика. Например : дължина на цяло число, алгоритъм за изчисляване на корен квадратен. Това означава, че изходните програми на дадена входна програма, получени при различни реализации на този език могат да ни бъдат еквивалентни програми. В повечето езици за програмиране се правят опити за минимизация на тази зависимост на езика от реализацията му.

3. **Свързване при писане на програма –** Писането на програма можем да определим като дейност по свързване, при която :

а) се дефинират програмни обекти
б) се дефинира последователност от промяна на характеристиките на така дефинираните програмни обекти. Можем да кажем, че програмирането е дефиниране(писане) на последователност от предефиниране.

Съществото при писане на програма е, че се използват възможностите, определени от реализацията на езика за програмиране, която ползва програмиста. Например от множеството на всички възможни величини(величина=(име, тип, текуща стойност)), той използва няколко като определя начина и последователността на промяна на текущата им стойност. Последното означава, че програмистът дефинира и възможите текущи стойности на величината, които би трябвало да се получат по време на изпълнение на програмата. Има ли връзка между казаното и понятието грешка?

Всяка една програма може да се разглежда като своеобразна заявка, която трябва да се „реализира". Следователно ако програмистът не познаваначина на реализиране на заявените от него във програмата му свързвания, то той не може да предвиди(достигне) такива качества на своята програма като бързина на изпълнение, компактност и вярност.
Пример :
var A : integer;
Ако програмистът допусне грешка, че свързването се реализира от транслатора като велечината А се занулява, то неговата програма ще притежава характеристиката грешка на програма(това има много силно влияние върху нервната система на програмиста).

4. **Свързване по време на транслация –** Свързване, което се извършва по време на транслация се нарича още **ранно свързване.** Примери са : определянето на типа на величина или определяне на мястото и размера й в оперативната памет. Колкото по-голяма част от допустимите в езика свързвания са равни, толкова изпълнимите програми са по-бързи, но по-неефективни по отношение на използвания обем оперативна памет. Освен това ранното свързване налага някои неудобства на програмиста. Например да работи със статични променливи. Езици реализиращи ранно свързване се наричат още ефективни

5. **Свързване по време на изпълнение –** нарича се още късно свързване. Например определяне на местоположението в оперативната памет на динамична променлива или свързване на променлива с текущата й стойност. Два важни подкласа късно свързване са:
    а) Свързване при активиране на програма;
    б) Свързване в произволна точка на програмата;

Очевидно е, че колкото по-голяма част от допустимите в език за програмирне свързвания са къси, толкова изпълнимите програми са по-бавни, но могат да бъдат направени(но само от добри програмисти) ефективни по отношение на използвания обем

оперативн памет. Последното предполага, че добрата програма се пише поне два пъти. Езици, предполагащи късно свързване се наричат още **гъвкави**.

6. Изводи

а) При бефиниране на даден език за програмиране обикновено се вземат предвид и начина на реализация на допустимите в него свързвания, т.е. този, който си измисля езика трябва поне принципно да знае начина на реализация на този език;

б) Замяната на ранно с късно свързване, при реализацията на език, не понишава гъвкавостта му(значително), но повишава ефективността му. Незначителни синтактични промени във версия на реализация на даден език твърде често означават промяна на времето на свързване;

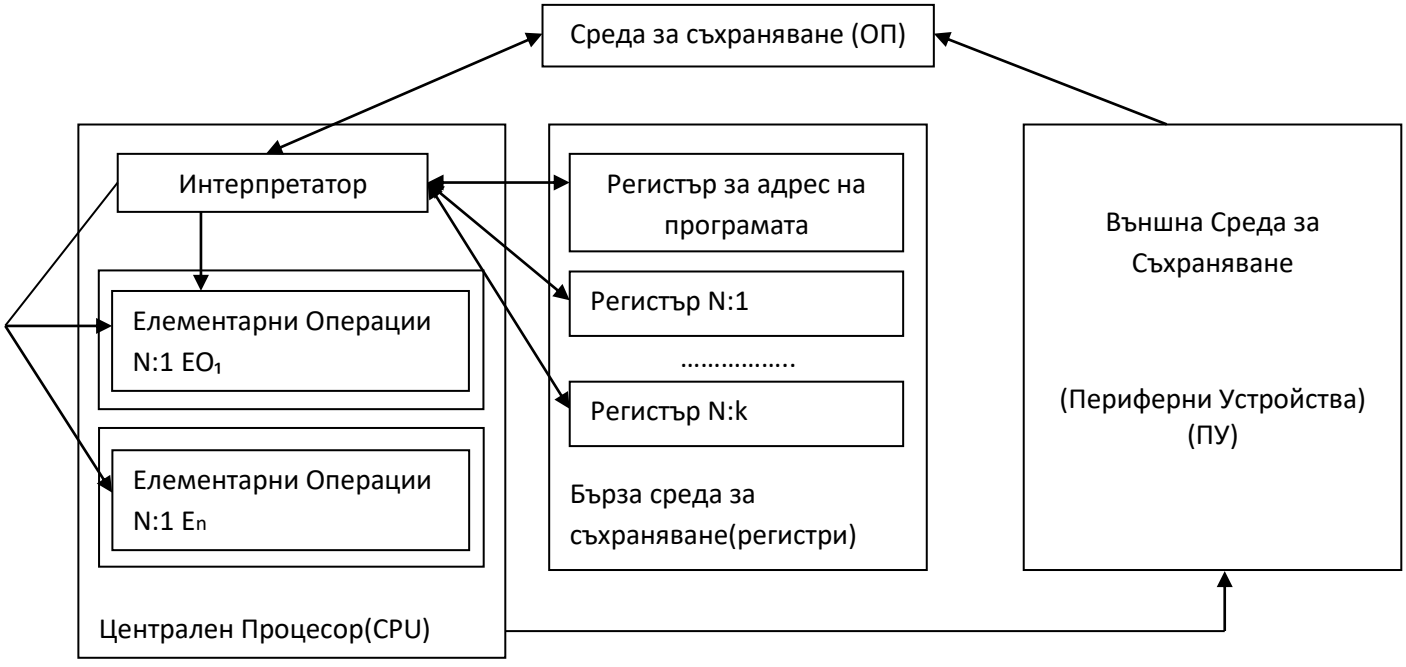
Например : var A : array [1…5] of byte;

var A : array of byte;

в) От гледна точна на курса лекции, съществени са ранното и късното свързване;

7.Базова ВИМ

Под базова ВИМ ще разбираме хардуерният аспект на ВИМ, който обозначаваме като компютър. Ще обсъдим БВИМ с минимални възможности, произтичащи непосредствено от принципите на програмно управление на фон Нойман. Типичната архитектура на такава машина има следния състав и структура структура:



Оперативната памет съдържа програмата и данните, необходими за нейното изпълнение. Обработката(изпълнението) на една програма се извършва от интерпретатора. Той избира(чете) последователно от ОП машинните команди на програмите. За всяка машинна команда интерпретатора извършва следните действия:

а) Разшифрова я;

б) Определя операндите й;

в) Активира заявената в машината команда елементарна операция с така оп определение параментри;

Всяка елементарна операция иозвършва определено действие върху придадените операнди и връща управлението на интерпретатора

Периферните устройства трябва да се разглеждат като специализирани процесори. Комуникацията им с централния процесор може да се реализира по различни начини, в зависимост от „интелигентността" на ПУ.

1. Данни

Логическия еквивалент на оперативната памет е крайна последователност от двойчни разряди, които обузначаваме като – bit. Тази последователност е разделена на части с равна дължина, които се наричат **клетки**. Всяка клетка се идентифицира еднозначно чрез поредния си номер в този вектор. Сега е прието клетките да имат дължина 8 бита и да се наричат байт, но това не е задължително. Всеки бит от ОП може да бъде в точно едно от две състояния, които ние обузначаване като 0 и 1. Данна наричамеконкретно състояние на битовете на дадена клетка. Следователно единствения тип данни са целите положителни двоични числа, които могат да се запишат(кодират) в една клетка. Разпознаването на такова двоично число като друг тип данни се извършва от операцията, която използва тази данна като аргумент. За да се дадат в тази архитектура минималния брой необходими за моделиране типове от данни се разрешава една операция да има достъп до няколко последователни клетки като тази последователност се разглежда като неделима данна, а достъпът до нея става посредством адреса на първата или последната и клетка. Максималния брой битове, до които има достъп едновременно ЦП определят компютъра като компютър с К-разрядна шина. Максималния брой битове, които може да обработва едновременно ЦП го определят като компютър с К-разряден процесор. Тъй като стойността на всеки бит може да се разглежда и като логическа стойност можем да кажем, че базовите типове данни са два:

а) Цели положителни двоични числа от даден диапазон;
б) Низове от логически стойности с фиксирана дължина;

Всеки бърз регистър е клетка, чиято размерност съответства на размерността на ЦП. Броят на общите регистри е малък, а предназначението им е да ускорят изпълнението на елементарна операция. Поради това обменът на данни между ЦП и бързите регистри е много по-бърз, отколкото между ЦП и ОП. Обикновено съвремените ЦП са едноадресни, т.е. на всяка бинарна операция единият от аргументите е в ОП, а другият е в бърз регистър, като резултатът се получава в бърз регистър. Регистърът за адрес на програмата е специален. През цялото време на изпълнение на една програма той съдържа адреса на машинната команда, която трябва да се изпълни след текущо изпълняваната машинна команда.

2. Елементарни Операции

Множеството от елементарните операции определя разнообразието от вградени типове данни на БВИМ. Типичното множество елементарни операции включва :

а) Аритметични операции над цели числа;
б) Аритметични операции над реални числа с фиксирана точка;
в) Аритметични операции над реални числа с плаваща точка;
г) Логически операции;
д) Операции за обмен на данни между/в двете среди за съхраняване - ОП и бързите регистри;
е) Операции за установяване на различни свойства във вградените типове данни;
ж) Операции за управление на последователността на изпълнение машинните команди;
з) Операции за управление на комуникацията с външната среда на съхранение;

Интерпретатора на ЦП поддържа особен тип данни, които ние наричаме тип на машинна комада или формат на машинна команда.

3. Управление на последователността от действия

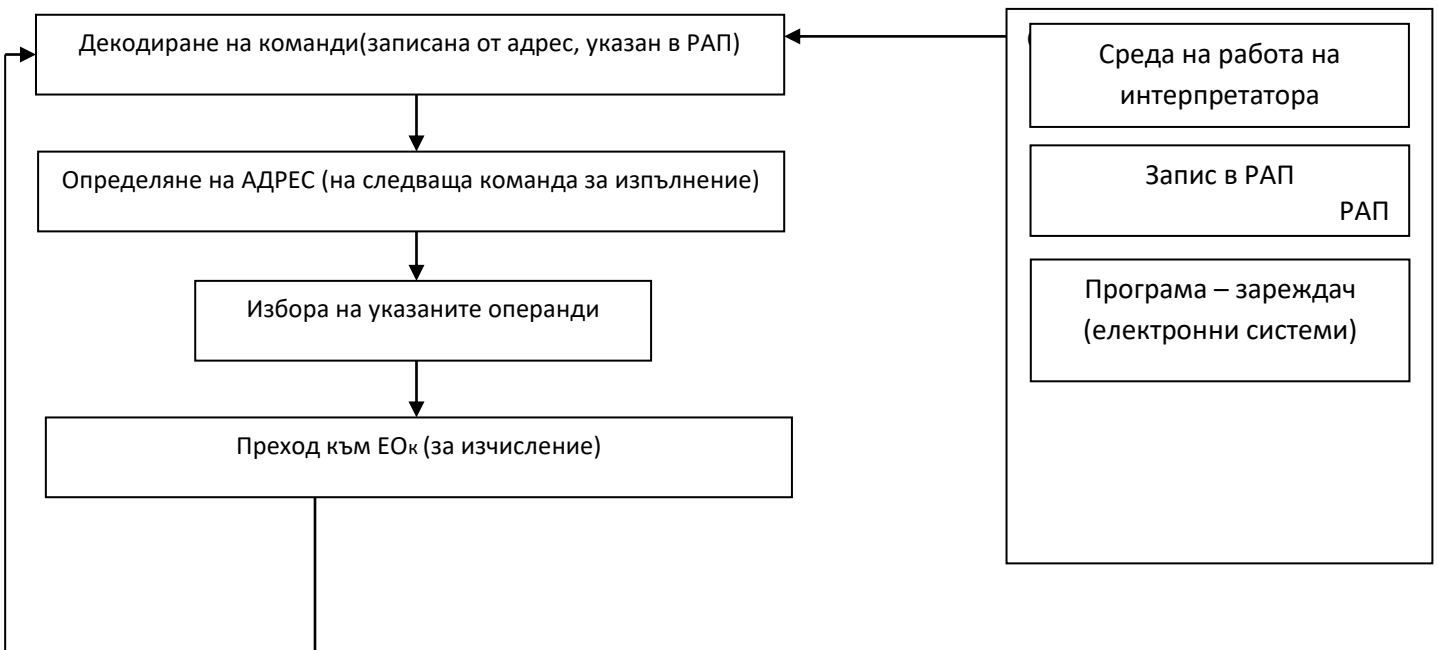
В БВИМ има два механизма за УПД – явен и неявен. Неявният механизъм е реализиран в интерпретатора и гласи, че машинните команди се изпълняват в последователност, определена от адресите в оперативната памет, където те са записани, т.е. първа се изпълнява командата, чийто запис в ОП има най-малък адрес. Това означава, че интерпретаторът може да определя адреса на следващата команда за изпълнение на базата на адреса и кода на текущо изпълняваната машинна команда. Явният механизъм за УПД включва две команди :

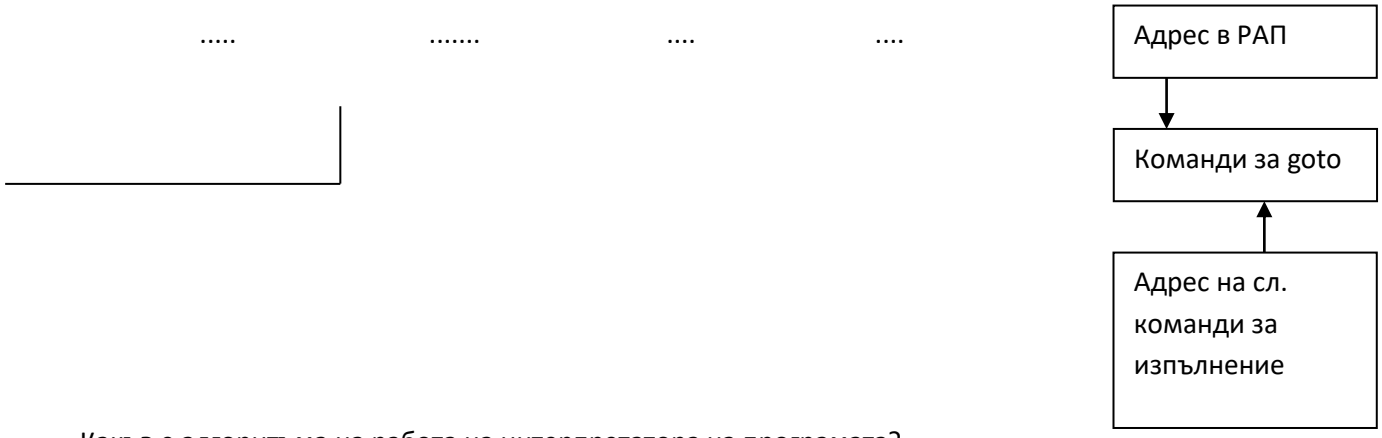
а) Команди за предаване на управлението при наличие на конкретно състояние на дадена данна. Тези команди се наричат още команди за условен преход. Ако конкретното състояние не е на лице се използва неявният механизъм.

б) Команди за безусловно предаване на управлението – команди за безусловен преход. Под предаването на управление разбираме, че се указва адреса, където е записана машинната команда, която ще трябва да се изпълни като следваща.

Тези два механизма се реализират посредством регистъра за адрес на програмата. Този регистър всякога съдържа адреса на командата, която трябва да се изпълни като следваща. Достъпа до този регистър може да се променя само от интерпретатора и елементарните операции, реализиращи явния механизъм. Алгоритъма на елементарната операцията, която ние нарекохме интерпретатор е следният :

Работа на Интерпретатор (от CPU):





Какъв е алгоритъма на работа на интерпретатора на програмата?

Може ли интерпретатора да се разглежда като подпрограма на ОС?

Задача : Създайте механизъм за управление на последователността на изпълнение на подпрограми, като :

а) Педполагате, че подпрограмите се изпълняват в съответствие с принципа на заместване;
б) Всяка подпрограма не приема и не връща данни, но може да активира друга подпрограма;
в) Не се допуска рекурсивно активиране на подпрограми;

Каква структура от данни е необходимо да се използва, за да може създадения механизъм да се използва за рекурсивно активиране на подпрограми.

4. Управление на Данни

Най-общо под управление на данни разбираме начините за описание на аргументите на машинна команда и съответстващите им механизми за достъп до тях. Има три места, където може да се намира операнд на машинна команда :

а) Непосредствено в машинната команда(той е недостъпен за запис);
б) В регисът;
в) В ОП;

Ако операндът е в регистър или ОП, то в машинната команда е записан съответен адрес. Обикновено адресите на регистрите са по-малки от адреса на първата клетка на ОП. В този случай се използват два механизма за достъп до данните (начини на адресиране) :

а) Директно адресиране – когато операнда се намира непосредствено на указания в командата адрес;
б) Индиректно адресиране – когато операндът се намира на адрес, намиращ се на указания в командата адрес;

Всеки адрес е вграден тип данни, за който има два начина за описание :

а) Абсолютно адресиране, когато адресът се описва като едно неделимо число;
б) Относително адресиране, когато адресът се описва чрез наредена двойка числа(база, отместване), чийто сбор дава абсолютен адрес.

Очевидно е, че съществува голямо, но крайно множество от начини за задаване на операнди. Това означава, че всяка елементарна операция има съответно множество от свои реализации в ЦП(по отношение на начините на адресиране на операндите), като на всяка реализация е съпоставен уникален числов код, който се явява име на операцията. Ясно е, че една и съща от гледна точка на извършваното действие елементарна операция съществува в много хардуерни варианти от гледна точка на начините за описание на операндите й.

5. Управление на Паметта

Защо е възможен неявният механизъм за управление на последователността от действия?

Такъв механизъм в БВИМ отсъства, т.е. програмата и данните са разположени на фиксирано място в ОП през цялото време на изпълнение на програмата. Ясно е една операция откъде черпи аргументи.

Как би изглеждала БВИМ, реализираща т.нар. парадигма за програмиране с името функционално програмиране.

6. Операционна Среда

Проблемът за общуването с Операционнта Среда се свежда до проблема за комуникация между две ВИМ. Съществуват много начини за комуникиране като тяхната реализация обикновено се нарича протокол. Най-простият случай на операционна среда е когато тя не съществува. Това означава, че елементарните операции на ПУ се разглеждат като подмножество на елементарните операции на ЦП, т.е. в системата от команди на ЦП съществува и множество от команди за управление на ПУ.

8.Базов ЕП()

Всеки мнемокод(автокод) е логическа експликация. Това означава, че между множеството от елементи на програмата, написана на мнемокод, и множеството от елементи на програмата, написана на машинен език, съществува взаимно еднозначно взаимодействие. Това позволява една БВИМ да бъде изучена в термините на съответния мнемокод. Менмокода се отличава от машинния език по три факора :

а) Цифровите кодове на машинните операции се заменят с буквени кодове, наречени още мнемонически;

б) Цифровите адреси на операндите се заменят с буквени такива или цифрови в друга бройна система;

в) Цифровото описание на непосредствените операнди се заменя с буквено цифрово, в друга бройна система, или с израз.

Как транслатора превежда мнемоническите обузначения на адреси в реални адреси на БВИМ?

Забележка: Ако този въпрос се падне на изпита, то ще трябва да го развиете във вид на ръководство за програмиста за изходния език на вашия транслатор. Ако в процесът на изучаване на изходния език сте създали такова ръководство, то ще ви бъде признато от изпитващия дори под формата на царски пищов.

9.Данни във входния език за програмиране

Когато говорим за данни трябва да различаваме три аспекта – логически, синтактичен и реализационен.

Под **логически аспект** разбираме абстрактната дефиниция на типа данни. Пример – масив се нарича множество от еднотипни величини, между чиито имена е установена строга наредба, посредством биективно изображение в свързано подмножество на пространството на целите числа.

Синтактичният аспект на данните определя начините за работа с данните в ЕП. Напрмер – горната дефиниция за масив позволява в имената на масивите да се използват цели числа (индекси на масива), чрез които се идентифицират елементите на масива.

Реализационният аспект определя начините за представяне на данните по време на изпълнение на програмата. Нашето обсъждане е концентрирано главно върху реализационния аспект на данните. По време на изпълнение на програма съществуват два класа данни –

а) Данни, определени от програмиста;
б) Данни, определени от системата(транслатора);

Последните се формират от транслатора (реализацията на езика) за служебни нужди и често програмиста не подозира за тях. Една от причините за наличие на такъв клас данните е късното свързване(по време на изпълнение). Например – ако входният език допуска в хода на изпълнение на една програма една променлива да полуава текущи стойности от различен тип, то по време на изпълнение на програмата ще бъдат необходими не само името, но и типа на текущата стойност на тази променлива. Това налага да се подържат допълнителни (системни данни) за описание на характеристиките на променливата, късаещи късното свързване. Такъв системен тип данни са **дискрипторите на данни**. Това са тип данни, чрез които се описват характеристиките на данните, определени от програмиста. Най-общо под представяне на данна в ОП се разбира двойката от следните два елемента -

а) Код – двуидно число, представящо текущата стойност на величината, т.е. данната в собствен смисъл;

б) Дескриптор – код, съдържащ допълнителна информация за първия елемент на представянето, т.е. за кода на данната. Тази допълнителна информация може да бъде относно името на данната, типът й (респективно начина на обработка на кода й), местоположението и размерите на кода по време на изпълнение и т.н. Можем да кажем, че дескрипторът съдържа полезна информация, късаеща не само времето на изпълнение.

Очевидно е, че всяко представяне на данни –

а) Може да не съдържа дескриптор, но е задължително да съдържа код;

б) Може да бъде разположено в последователни клетки на ОП или „разхвърляно" в ОП. Казано на професионален език представянето на данна в ОП може да бъде последователно или свързано. За да избегнем тафтологии вместо термина представяне на данни в ОП ще използваме термина **позиция на данна в ОП**. Ясно е,че всяка операция, обработваща данна получава достъп до нея, чрез адреса на позицията на данната. Една операция се нарича **специфична** ако алгоритъмът й не използва дескрипторите от позициите на нейните аргументи и резултати. В противен случай операцията се нарича универсална, тъй като по същество изпълнява различни алгоритми в зависимост от съдържанието на дескрипторите.

Какви са операциите на БВИМ? –

Декларация наричаме всяка инструкция на ЕП, която описва характеристиките на данните по време на изпълнението. Следователно на всяка декларация от програмата съответства позиция по време на изпълнение.

Задача – Да се дефинира позицията на даннта, определена от декларацията.

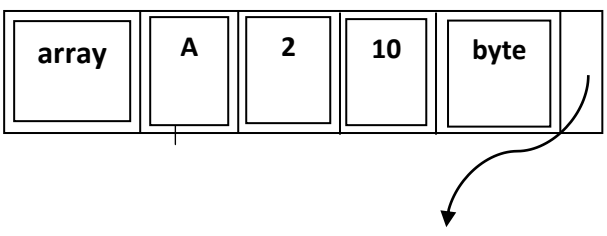
var A : array [2…10] of byte;

Решение : За да решим задачата трябва да „декодираме" характеристиките, описани чрез тази декларация. Те са -

1. Името на величината - A
2. Тип на величината - array
3. Тип на елементите - byte
4. Брой на елементите - 9
5. Множество от индекси, чрез които се именува елемент - {2,3,4,5,6,7,8,9,10}
6. Момента от времето, когато се създава позицията - при визане в съответния box.
7. Момента от времето, когато се унищожава позиция - при напускане на box.

Тези характеристики трябва да се разпознаят от транслатора и на тяхна база транслатора да генерира множество от машинни команди, в изходната програма, които създават и освобождават позиция със следната примерна структура :

Дескриптор



.....

Очевидно е, че декларациите позволяват универсалните операции на входният език да се транслират в специфични операции на изходния език.

Въпрос : Върху кои други аспекти на ВИМ влияят декларациите?

Казваме, че входният език допуска статична проверка на типа, ако е възможно универсалните му операции да се транслират в специфични. Алтернативата е динамична проверка на типа, която се извършва от универсалните операции по време на изпълнение на програмата. Очевидно декларациите са средство за повишавате на ефективността на езика.

Въпрос : Съществува ли и друг начин за деклариране на характеристики освен обсъдения вече явен начин (декларация)?

Различаваме прости и съствни типове данни.

Задача : Класифицирайте типовете данни в избран от вас език за програмиране?

Централно място при работа с данни заема операцията **достъп до елемент на съставен тип данни.** С този термин се обозначават две операции :

а) Достъп до стойност обузначават, чрез A[2] в x:=A[2]+5;

б) Достъп до позиция, чрез A[2] в x:=A[2]-5;

Очевидно достъп до позиция е по-универсална, тъй като на нейна база можем да реализираме и достъпа до стойността. Поради това тези две операции ще наричаме с общото има **достъп(до данни)** и ще го разбираме под аспект достъп до позиция ако не уточнен вида на достъпа.

Ще правим разлика между операцията достъп и операцията позоваване.

За да се изпълни операцията достъп, обозначен чрез A[2] първо трябва да се идентифицирапрограмният обект, който в този момент носи името A, т.е. това означава да се определи адреса на дескриптор и този адрес да стане един от аргументите на операцията достъп. Този етап на определяне на адреса на дескриптор наричаме операция **позоваване на данна.** Тази операция има отношение към аспекта управление на данните на ВИМ.

Как се дефинира операцията достъп в език за програмиране?

Какво означават термините интерпретируеми структури от данни и разширяеми структури от данни?

10.Операции във входния език за програмиране

Когато говорим за операции трябва да ги обсъждаме в три аспекта – логически, синтактичен и реализационен.

Логическия аспект означава, че операцията се разглежда като алгоритъм на абстрактни типове от данни. Например – алгоритъм за изчисляване на корен квадратен от някаква величина.

Синтактичният аспект е начинът за описание на операциите в ЕП. Ийма два начина за синтактично описание на операция :

а) Като знак в програмата, което означава, че програмистът не е длъжен да знае алгоритъмът на нейното изпълнение. Например – a+b, sin(x).

б) Като подпрограма на програмата, което означава, че програмистът описва или има в явен вид алгоритъма на изпълнение на операцията.

Необходимо ли е разработчикът на транслатор да знае алгоритъма на всяка допустима в езика операция? – Да

Реализационният аспект е начина за описание на операция в термините на БВИМ. Оченидно е, че част от операциите на входния език за програмиране, които се обозначават чрез знаци имат съответни алтернативи в БВИМ. Например – аритметичните операции. Но друга част трябва да се реализира като последователност от команди на БВИМ. Например – корен от **x**. От казаното следва, че допустимите в език за програмиране операции могат да се разделят на три класа –

а) Вградени(имат аналог в БВИМ)

б) Примитивни(обузначават се чрез знак, но нямат аналог в БВИМ)

в) Подпрограми

От казаното следва, че обсъждането на всяка операция трябва да бъде съобразена с реализационния аспект поради следните причини :

а) Практически не е възможно да се опишат формално всичките възможни комбинации от стойности на входните аргументи;

Например – var A,B : byte;
A+B;
255+2=257>255?
+(255,2) – грешка

б) Операцията може да използва неявни аргументи;

в) Операцията може да предизвиква странични ефекти (промяна на стойността на групи величини, които не са описани като неин резултат);

г) Самоизменение на операцията и като код, и като собствени данни;

Очевидно всяка преведена програма съдържа както операции, определени от програмиста, така и операции, определени от системета. Наличието на последните е неизбежно. Защо? – има данни, определени от системата => има и операции, определени от системата.

От гледна точка на обектите, върху които се полагат операциите допустими в даден език се делят на две групи :

а) Итерации върху дефинирани от програмиста типове данни – вградени, примитивни, подпрограми;

б) Операции за управление на паметта. Те са само част от операциите, определени от системата;

Трябва да се отбележи, че реализацията в транслатора на част от операциите от първата група изисква използвани на операции от втората група. Очевидно при разработването на транслатор трябва да се започне с операциите за управление на паметта и другите операции, определени от системата.

наричаме елем

Обзор на операциите :

I. **Елементарни операции**
    1. **Аритметични операции**, като различията между езиците за програмиране по отношение на аритметичните операции са основно във възможността за статична проверка на типа;
    2. **Операциите за сравнение (предикати)** - <,>,= ... ;
    3. **Логически операции** - . , | , отрицание ... ;
    4. **Операции за преобразуване на тип** – тези операции се използват и неявно като операции определени от системата;
II. **Неелементарни операции**
    1. **Присвояване** – операция със страничен ефект, защото променя стойността на един от своите аргументи. Конфликтът се получава ако аргументите й са от различен тип; Въпрос – По колко начина може да се разреши този конфликт? - По два начина.
    2. **Създаване на структура от данни (СД) (create)** ;
    3. **Включване на елемент в структура от данни (add to);**
    4. **Унищожаване на структура от данни (dispose);**
    5. **Изключване на елемент от структура от данни (delete from);**

Операциите 2 и 3 трябва да се реализират така, че след тяхното изпълнение да са възможни операциите позоваване и достъп. Операциите 4 и 5 са опасни. Некоректното им изпълнение поражда проблемиете, наричани **висящ указател и боклук**.

Задача – Дефинирайте проблемите висящ указател и боклук.

6. **Поставяне в образец** – търсене на подструктура с определени качества и евентуалната ѝ замяна с друга подструктура. Пример – substr('ABV','B','..'). Ще се получи : 'ABV' -> 'A..B'
7. **Операции, дефинирани от програмиста (подпрограми**); Въпрос – какъв е общо приетият начин за обозначаване на изпълнението на подпрограмите? Префиксен запис на операцията (знак на операцията последван от списък на нейните аргументи, разделени със запетайка, написани в скоби);
8. **Операции в програмите –** изпълнение на програма, транслация (разглеждаме я като операция на програма);

11.Управление на последователността от действия(УПД)

Управление на Последователността от Действия(УПД)

Изхождайки от йерархичните връзки между елементите на ЕП, механизмите на УПД се делят на три групи :

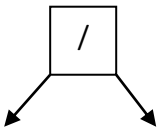
1. Механизъм за УП от оператори в изрази;
2. Механизъм за УП на изпълнение на опеаторите в езика;
3. Механизъм за УП подпрограми;

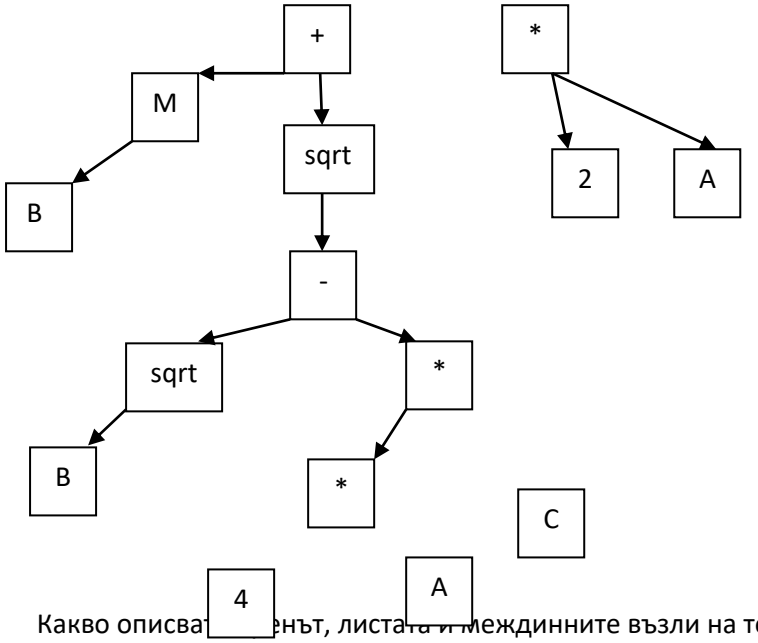
Тези механизми могат да бъдат явни или неявни. Неявни – механизми, които съгласно дефиницията на езика действат във всички случаи, докато програмиста не ги замени с явни такива. Например неявният механизъм за ПИ може да бъде променен от механизмите : последователно изпълнение, разклонение, повторение. Очевидно за задаване на явен механизъм в ЕП трябва да съществуват съответни изразни средства. Пример за явни инструменти за смяна на тези механизми са скобите в изразите или оператор GOTO и етикетите на ниво оператора.

Обикновено количеството допустими механизми за УПД в даден ЕП и тяхната съгласуваност с данните и операциите на този език се използват като критерий за класифициране на езика за прогрмиране като ЕстественЕ. Защо езикът Pascal се класифицира като естествен език? Защо Pascal е разработен за първонално обучение на студенти по програмиране?

I. Управление на последователността от изрази

Основният механизъм за УПД в изрази се наича Функционална Композиция. За да се построи е необходимо да се зададе главна операция и операнди, които също могат да се получават в резултат на изпълнение на операция. По този начин ФК Придава на израза структура на дърво.





Какво описват елементът, листата и междинните възли на това дърво? Очевидно е, че дървотот внася яснота, но не е ясен реда на изчисляване. Поради това трябва да се даде и правилото за обхождане на дърво. Това правило се задава чрез синтаксиса на езика като се използват префиксен, инфиксен или суфиксен начин на запис на операциите в израза. Освен това в реализацията се фиксира :

а) Еднородно парвило за изчисление, съгласно което всички операции се разглеждат или като специфични или като универасални.

б)Разрешението или забраната за ползване на функции, реализиращи странични ефекти;

в) Начина за реагиране на грешка, възникнала при изпълнение на Елементарни Операции;

Какво представляват приоритетите и скобите в изразите?

II. Управление на последователността на изпълнение на операторите

Всеки такъв механизъм трябва да бъде приложим в термините на механизма на БВИМ. Директен аналог в ЕП е апаратът , но в ЕП могат да съществуват три варианта на оператор GOTO в развисимост от дефиницията на понятието етикет. Различаваме следните етикети :

1. Етикети като локални синтактични указател, чиито машинен еквивалент (адрес) се определя по време на транслацията.
2. Етикети като данни, изчислими по време на транслацията.
3. Етикети като данни, които се изчисляват по време на изпълнението.

Как се превеждат трите разновидности на оператора GOTO?

Тъй като явното използване на оператора води до лош стил на програмиране се наложи структурното програмиране, в което се използват конструкциите разклонение и повторение.

Задача Преведетепознатите ви конструкции за повторение и разклонение в конструкции на БВИМ.

За да се справим с тази задача трябва да си начертаем съответните блок-схеми.

III. Управление Последователността от изпълнение на подпрограми

Най-използвания механизъм от този вид се базира върху така нареченото правило за копиране – Ефектът от изпълнението на оператор за изпълнение на подпрограма е същият както ако този оператор преди изпълнението си е заменен с копие на тялото на подпрограмата, в което са направени съответните замени на формлните с фактическите параметри. За реализацията на този механизъм в БВИМм обикновено има команда наричана преход с възврат. Тя предава управлението във входната точка на подпрограмата и записва на определено място точката на възврат. Последната команда на подпрограмата (RETURN) чете от това място точката на възврат и предава там управлението. Ясно е, че записът на точката на възврат е някъде в тялото. Правилото за копиране съдържа неявно пет ограничения, чието отхвърляне води или поражда други механизми на УПоИ на други подпрограми. Тези ограничения са :

1. Подпрограмите не могат да бъдат рекурсивни. Ако премахнем ограничението получаваме рекурсивно активиране.
2. Необходим е явен оператор за изпълнение на подпрограни.(CALL/GOSUB). Ако премахнем ограничението получаваме прекъсване, събитийно програмиране.
3. При всяко активиране подпрограмата се изпълнява до логическия си край. Ако премахнем ограничението получаваме т.нар. съпрограми.
4. Последователността на изпълнение на подпрограмите е единствена. Ако премахнем ограничението получаваме т.нар. задачи и паралелно програмиране.
5. Незабавно предаване на управлението в точката на възврат. Ако премахнем ограничението получаваме т.нар. планируеми подпрограми.

Трябва да правим разлика между определение на подпрограма и активация на подпрограма.

Определението на подпрограма служи за образец, по който по време на изпълнение на програма се създава активация. Определението се изготвя от транслатора, съгасно нейната дефиниция и се състои от :

а) Постоянна част, съдържаща онези програмни елементи, които са неизменни по време на изпълнението.

б) Шаблон на активизационния запис, съдържащ онези програмни елементи, които зависят от момента и мястото на активиране на ПП.

Когато ПП трябва да се изпълни първо се създава активизационен запис (АЗ). Този АЗ и постоянната част формират активацията на ПП (АПП). Очевидно в даден момент могат да съществуват няколко активации на дадена ПП. Които ще се различават помежду си само по АЗ. Въпрос : Кои програмни елементи на ПП формират шаблона на АЗ? Задача : дефинирайте понятието активация на ПП в мрежов режим на работа.

12.Управление на данните

Всяка програма преписва операции не над конкретни данни, а над множества от ДС, т.е. над типове от данни. Поради това **позоваванията на данните** трябва да бъдат в преоладаващата си част косвени, т.е. да се използват имена. В ЕП от високо ниво се разрешава както едно име да се използва многократно, т.е. в различни моменти от време с него да се обузначават различни програмни елементи, така и един и същи програмен обект да се именува с различни имена в един и същи момент от време.

Въпрос : По какво се отличава начина на именуване в ЕП от високо ниво от този на езика на БВИМ?

Централно място в УД заема въпросът за смисъла на имената. Този въпрос гласи – Кой е програмният обект, който в дадения момент и на даденото място се именува с този идентификатор? Следователно централна тема на нашето обсъждане е операцията **позоваване на данни**. С терминът асоциация ще обузначаване наредената двойка (идентификатор, дескриптор), без да се интересуваме от начина й на представяне. Все пак за конкретност допускаме, че асоциацията е запис в релационна талица с два домейна – име и дескриптор. Основните операции за УД, които имат отношение към асоциациите са пет :

1. Именуване – създаване на асоциация.
2. Разименуване – унищожаване на асоциация.
3. Активиране – разрешение за ползване на асоциацията от операцията позоваване.
4. Деактивиране на асоциация – забранява се използването на активация.
5. Обработка на позоваването – търсене в таблицата с имената сред активните асоциации на идентификатори.

Начинът на търсене е от съществено значение за идентифицирането на дадена асоциация като тази, която търсим. Множеството от активни, в даден момент, асоциации ще наричаме среда на позоваване (СП). По същество средата на позоваване се определя от операциите 1, 2, 3, 4. Съвкупността от правилата за извършване на тези 4 операции се обузначава като област на действие (на име). В зависимост от това дали тези правила отразяват процеса на изпълнение на програма или процеса на транслация се различават съответно динамична и статична област на действие.

Защо множество от правила наричаме област на действие ?

От гледна точка на програмиране се различават локална среда на позоваване и нелокална среда на позоваване. Втората е изградена от глобални и нелокални асоциации. Могат да се направят следните изводи :

1. Областта на действие произтича от механизма за управление на последователността изпълнение на ПП, а в някои езици за програмиране зависи и от правилата за дефиниране на ПП.
2. Динамичната област на действие предполага наличие и използване на среда за позоваване в изпълнимата програма.
3. Практически областта на действие се фиксира строго от разработчика на транслатора, но отсъства в явен вид в ръководствата по езика.

Задача – Нека ПП P активира ПП Q, а Q – R. Отговорете на следните въпроси :

1. Каква начална локална среда се установява в Q при вход от P?
2. Какво става с локалната среда на Q, когато Q активира R?
3. Каква ще бъде локалнат среда на Q, когато получи обратно управлението от R?
4. Каква ще бъде локалната среда на Q, когато върне управлението на P?

Въпроси:
Какви видове формални параметри може да притежава ПП?
Колко начина съществуват за предаване на параметри? – по стойност, по адрес, по име.
Определяне вида на формалния параметър и начина за предаване параметър?

13.Управление на паметта

УП е една от основните грижи както на програмиста, така и на разработчика на транслатора. Основните обекти, за които е необходима памет по време на изпълнение са следните :

1. Транслираната програма
2. Системните програми по време на изпълнение
3. Дифинираните от програмиста данни
4. Памет за механизма за УПД
5. Среда на позоваване
6. Временна памет за междинните резултати при изчисляване на изрази
7. Временна памет за предаване на параметри при активиране на подпрограми
8. Буфери за входно-изходни операции
9. Системни данни
10. Други

Съществуват три основни аспекта за УП :

1. Начално разпределение на паметта.
2. Утилизация на паметта – откриване на заделена, но неизползвана още памет.
3. Уплътнение на паметта .

Тези аспекти би трябвало да се изучават по дисциплината Операционни Системи.

Тъй като транслаторите са системни програми, можем да считаме, че разработчикът на транслатор ползва додтъпният в дадена ОС механизъм за УП.

14.Операционна среда

Определихме понетието ОС на една програма като множество от програми, с които тя си комуникира посредством съобщения.

Задача. Опрделете начините на комуникация с Операционната Среда на програма, написана на познат ви език за програмиране.

II.Техники на реализация на транслатори

Основното предназначение на синтаксиса е да предостави система от обозначения за обмен на информация между програмиста и изпълнителя на неговата програма.

Общия стил на синтаксиса се определя от множеството избрани синтактични елементи. Ще обсъдим тези от тях, които са най-характерни за ЕП от високо ниво.

1. Азбука - Множеството от знаци, допустими в ЕП наричме азбука. Трябва да се прави разлика между азбуката на езика и множеството от знаци(символи), допустими за входно-изодните операции на този език. Обикновено тези две множества имат сечение, но азбуката съдържа и низове от символи, които за езика са неделими. Освен това знаците на азбуката се класифицират в непресичащи се множества като букви, цифри, знаци за операции, ограничители, разделители, служебни думи. Много често в официалното описание на езика тази класификация не е дадена, но тя съществува неявно в множеството от металингвистични формули. Поради това първата и основна задача на разработчика на транслатор е дс направи тази класификация, което означава да дефинира подграматика с аксиома : <АЗБУКА>::=??...
2. Лексикален фонд – Лексемите са средството за обозначаване на програмни обекти, т.е. лексемите може и трябва да се разглеждат като имена. Ако едно име описва или съдържа в себе си стойността на програмния обект, то се нарича литерал. Например : 3.14. По същество всеки литерал представя стойност на вграден в езика тип данни. Следователно синтаксисът на всеки литерал го съотнася еднозначно към даден тип данни. Върху имената на собствен смисъл също могат да бъдат наложени ограничения с цел типизиране на програмните обекти, които те именуват . Например : име на подпрограма, име на реална величина и т.н., но това не е задължително. Разработчикът на транслатор трябва да дефинира строго подграматика с аксиома : <ЛЕКСЕМА> ::=??..., като се базира на граматиката азбука и с всяко понятие на тази граматика да свърже допълнителна семантична информация. Например : диапазоните на величините от чиалов тип или максималната дължина на низовете. По същество това е допълнителна информация за типа и характера на свързванията и времената на свързванията. В тази граматика могат да се появят и недефинирани понятия. Това произтича от рекурсивния характер на мета език. Обикновено смисълът на единтификаторите на имената се определя от контекста, в който те ще бъдат разпознати, но има и такива имена, чиято функционалност е строго регламентирана. Такива групи лексеми са :

   а) Ключови куми – обикновено те са фиксирана част – if , while, then …

   б) Резервирани куми – ключови думи, които не могат да се използват за други нужди(не могат да бъдат предефинирани).

   в) Шумови думи – думи, еквиваленти на коментар, чиято цел е да облекчат четенето на програмата – go **to**.

   г) Коментар – лексема, която се игнорира от транслатора.

   д) Разделител – особена лексема, която се използва, за да ограничава лексемите – интервал.

   е) Ограничител – обособява програмни обекти над лексените - ;

Обикновено някои от особ ените лексеми са част от азбуката на езика.

3. Операции

Знаците за обозначаване на операции трябва да се разглеждат като част от лексикалния фонд на езика, която ние обособяваме поради спецификата на тяхното използване. Обикновено в езика за програмиране с един знак се обозначава универсална операция. Например чрез + се обозначават – събитане , обединение на множества и конкатенация на изрази. Освен това като се изхожда от абстрактните дефиниции на операциите, обозначавани чрез даден знак, с него неявно се свързва и приоритета на тези операции. Тази допълнителна информация е достъпна неявно в дефиницията на понятието израз, но е добре тя да е дефинирана в явен вид от разработчика.

4. Изрази

Всеки израз е правило за изчисляване на стойност. Обикновено типът на резултата се преписва като характеристика на самия израз. Разработчика на транслатор трябва да познава много добре метаезика, защото механизма за управление на последователността от действия в изрази, който той трябва да реализира е дефиниран неявно чрез :

а) Понятията, които се използват за дефиниране на понятието израз;
б) Последователността на изброяване на алтернативите при дефиниране на понятията от подточка а).

Пример – металингвистичната формула

<ИЗРАЗ> ::= <прост израз>|<прост израз><мултипликативна операция><операнд>

Означава, че изразът се изчислява от ляво на дясно. Трбва да се отбележи, че в този пример се използва неявното правило за четене, приложимо както в метаезика, така и в програмата.

5. Декларации

Средствата, чрез които се описват реалните свързвания, които разработчикът трябва да реализира. Делим декларациите на две групи – декларация на програмен обект и декларация на операция. Нй-очевидния пример за декларации на операция са декларации на подпрограмата. Декларациите могат да бъдат автоматни и съставни. Като съставните се разглеждат като списък от автомати. Обработката на командна операци означава, че трябва да се извършат три дейности(откого кога и как).

а) Генерация на позиция
б) Генерация на асоциация в съответната среда
в) Активиране на асоциацията

В зависимост от областта на действие на даден ЕП за различните типове програмни обекти може да се извърши различно подмножество от тези три дейности, но генерирането на асоциация е задължително във всики случаи. Ако характеристиката, която се преписва в атомарна декларация е съставна (например var A : array [1…5] of array [2…7] of byte) е добре съставните характеристики да се разбият на атомарни такива чрез служебни идентификатори.Примера ще се приведе в следния вид :

type #1 = array [2…7] of byte;
var A : array [1…5] of #1;

Този преход се разглежда като подпроцес.
Очевидно е, че обработката на декларация на подпрограма се заключава в рекурсивното изтълнение на операцията транслация.

6. Прости Оператори

Под прост оператор разбираме:

а) Оператор за вход-изход
б Оператор за привсвояване;
в) Оператор за безусловен преход;
г) Оператор за акативиране на подпрограма;

Практика е операторите за вход-изход да се заменят от подпрограми, чиито имена могат да бъдат предефинирани. Обработката на такъв прост оператор означава да се извърши(в случай на интерпретация) или да се генерира последователност от операции, извършваща(в случай на компилация) преписаното от оператора действие (върху заявените операнди в съответсвие с областта на действие на дадения ЕП). Очевидно е, че ако една програма е изградена само от прости оператори, то тя ще бъде или линейна програма, или зациклена.

7. Съставни Оператори

Съставните оператори са явния механизъм за управление на последователността от изпълнение на оператори. Те могат да се разглеждат и като универсални метаоперации в езика за програмиране. Съществуват две основни конструкции на съставните оператори, наречени – разклонение и цикъл. Техните варианти и интерпретации лесно се превеждат в термините на последователност от прости оператори, етикети и условен оператор за преход. Следователно можем да считаме, че единствения съставен оператор като необходимост е условния оператор за преход. Разглеждн като конструкция, която може да има разнообразни проявления в езиците за програмиране. Обикновено дефинираме съставния оператор като оператор, чиито аргументи могат да бъдат също оператори и то съставни. Това означава, че на всекиоператор в една програма може да бъде съпоставено т. нар. ниво на вложеност. От тази гледна точка под оператор на програма разбираме всеки оператор от нулево ниво на вложеност. Очевидно е че, че в транслатора се обособи универсална рекурсивна операция транслация на оператор, то обработката на съставен оператор от ниво 0 няма да се разрешава от обработката на прост оператор от това ниво. Този поглед върху съставните оператори ни позволява да разглеждаме всяка програма като последователност от декларации и оператори от ниво 0. В съвременните езици за програмиране имаме възможност да обособяваме такава възможност в т.нар. съставен оператор или блок. Ако такъв оператор е именуван, то той се причислява към подпрограмите. Важното е, че чрез това понятие блок се дефинира неформално областта на действие на ЕП. Това правило гласи : Всяко име е валидно само в рамките на блока, където е дефинирано. Това означава, че всяка програма неявно се счита за именуван (блок). Следователно всеки допустим тип програмен обект трябва да бъде съотнесен с това правило, за да могат да се получат сумарните условия за реализацията на операцията позоваване(което е задача на реализатора).

8. Област на Действие (ОД)

Различаваме два вида област на действие:

а) Статична ОД, когато отношението вложеност на два блока се определя от гледна точка на разположението им текста на програмата(в Pascal областта на действие е статична).
б) Динамична ОД, когато това отношение на вложеност се определя от гледна точка на разположението им по време на изпълнение съгласно правилото за копиране.

Очевидно за всеки тип програмен обект се определя едно от тези две правила. Това означава, че :

а) Това се оказва в описанието на езика (връзката е твърда);
б) Или езика за програмиране предлага конструкции за указване на предпочитано правило както по време на транслация, така и по време на изпълнение.

Задача. Опишете предназначението и възможния начин на обработка на други синтактични елементи в съвременните ЕП като : клас, обект, интерфейс, модул, библиотека и т.н.

15.Сентактични елементи на език за програмиране

Всяка програма е краен низ от знаци. Казваме, че програма е правилна, ако този низ е породен от конкретна граматика. Ние можем да превеждаме само правилни прогрми, следователно задача в процеса на транслация е да се провери дали низът е породен от дадена граматика. Низът, т.е прогрмата е породена от дадена граматика ако съществува дърво за което е в сила :

a) Коренът е аксиома на граматика

b) Листата му са знаците на низа

c) Междинните възли съответстват на нетерминалне знаци на граматиката,

Това дърво наричаме синтактично дърво на програма(СД), а процесът на построяването му – синтактичен анализ на програма.

Съществуват два класически начина за изграждане на синтактичното дърво(СД) :

a) низходящ(отгоре -надолу) – когато дървото се строи от корена към листата.
b) възходящ(отдолу нагоре)- когато дървото се строи от листата към корена.

Аргументите на алгоритъма за изграждане на СД са два :

a) програма(низ)
b) граматика

Представянето на програмата е низ от знаци. В този низ знаците за делена на програмата на редове са заменени с разделители(с особени знаци) или пък са описани в граматиката като **разделители**.

Втора стъпка -Трабва да фиксираме представянето на граматиката. Приемаме,че граматиката е от тип LR(1) и налагаме следните съглашениея :

a) На всеки нетерминален символ съпоставяме по единствен начин цяло положително число заградено от <>глови скобки, като на понятието програма съпоставяме <програма> или <0>.
b) Азбука на граматикта наричаме съвкупноста от терминалните и знаци и знаците определени в подточка а).
c) Във всяка металингвистична формула заместваме понятието със съответните им знаци по дефинираната й азбука, като по този начин всяка формула става низ.
d) Сортираме получените низове във възходящ ред.
e) Всяка формула, т.е всеки низ който съдържа повече от една алтернатива заместваме със съответния брой формули без алтернативи, като спазваме последователността на алтернативите.

Полученото множество от низове записваме във таблица с две колонки като понятието отляво от знака по определение записваме в първата колона, а остатъкът от низа без знакът по определение в дясната колона.

Лява колона	Дясна колона
<0>	<……>
<0>	<……>

С низът от з придвижва указателят

Предполага знак в този низ.

Отличителн стъпка разпознава т правила на граматикат някаква синтактичн а чрез целенасоче се реализира по много начи

1. Реализираме логическа функция(предикат) която сръвнява текущият знак от низа на програмата със своя входен аргумент. Нека кръстим **функцията function EUQ(): Boolean**. При открито съвпадение придвижва Pnt напред.
2. На всеки ред от таблицата която описва граматиката, съпоставяме предикат, която връща стойност истина ако разпознае надясно от Pnt подниз породен от правилото записано в дясната колона на този ред. Алгоритъмът на този предикат е линеен и представлява последователност от извиквания на сродни на него предикати и на функцията EUQ. Ако резултатът е лъжа е необходимо указателят Pnt да бъде

възстановен със стойността си при влизане в гози предикат. Правилото за образуване на името на такъв предикат е P_<номер на понятие>_<номер на алтернатива>

3. На всяко понятие дефинирано, чрез повече от една алтеранитива, т.е няколко последователни реда от таблицата с еднаква колона се съпоставя предикат с името **P_<номер на понятие>**. Алгоритъмът на този предикат е прост – последователно активиране на предикатите реализиращи алтернативите докато не се поличи резултат истина или не си изчерпат алтернативите. Мерки за възстановяване на първата не са необходими.
4. Ако изпълнение на предиката P_o върне стойност истина или true анализираният низ е правилна програма по отношение на дадената граматика. Този начин на реализация на низходящият граматичен разбор притежава следните достойнства :

a) Разпознавателят на програма се модифицира много лесно ако към граматиката се добавят или отстраняват правила, това означава че ЕП може да бъде развиван лесно.
b) Синтактичното дърво не се строи в явен вид, но е възможно във всеки предикат да се добавят операции за изграждането му като тяхното използване е опционално.
c) Чрез донастройване на така определените предикати лесно се преминава от разпознаване към интерпретация и последваща компилация.

I. Възходящ анализ

Най-късият ляв подниз на програмата, който е непосредствено приводим към синтактична категария(т.е да е в дясната колона на таблицата) се нарича лява основа.Същността на възходящият анализ се заключава в търсене на основа и замяната на тази основа с съответния и нетерминален знак(т.е това е непрекъснато скъсяване на низа отляво) докато се получи нетерминалният знак на аксиомата на граматиката или пък се окаже, че няма основа. Очевидно е, че ефективността на разпознавателя от ефективността зависи от ефективността на алгоритъма за търсене на основи. Една примерна проста реализация на този възходящ анализ се заключава в следното : Изгражда се рекурсивен предикат, чиито глобални аргументи са низа на програмата и таблицата на граматиката. Алгоритъмът му е следният :

a) Търси се последователно низ, определен в дясната колона на таблицата на граматиката , който съвпада с началото на низа на програмата.
b) Ако не бъде открито съвпадение се връща стойност лъжа.
   Ако се открие съвпадение, т-е отново, то основата се замества със знака от лявата колона на таблицата и предикатът се активира отново рекурсивно :
   - Ако рекурсивното активиране върне стойност истина, то тя се връща и като резултат от работата на предиката.
   - Ако рекурсивното извикване върне стойност лъжа, то се възстановява състоянието от преди рекурсивното извикване и се продължава търсенето на основа със следващият низ(ред) от таблицата.

c) Ако активирането на тази процедура върне резултат истина, то програмата е правилна.

След прилагането на някои програмистки техники и хитринки може да се окаже, че възходящият анализ е по-бърз в сравнение с низходящият.Проблемът тук е , че за да се реализира преходът разпознаване, интерпретация , компилация е необходимо да се изградят същите предикати както при низходящият анализ(няма да бъдат предикати, а процедури) и да се натоварят със същите функции(по отношение на превода със изключение на дейността разпознаване). За нашите нужди, ще използваме низходящата разпознаване.

16.Синтактичен анализ на програма(разпознаване)

Всяка програма е краен низ от знаци. Казваме, че програма е правилна, ако този низ е породен от конкретна граматика. Ние можем да превеждаме само правилни прогрми, следователно задача в процеса на транслация е да се провери дали низът е породен от дадена граматика. Низът, т.е прогрмата е породена от дадена граматика ако съществува дърво за което е в сила :

d) Коренът е аксиома на граматика

e) Листата му са знаците на низа

f) Междинните възли съответстват на нетерминалне знаци на граматиката,

Това дърво наричаме синтактично дърво на програма(СД), а процесът на построяването му – синтактичен анализ на програма.

Съществуват два класически начина за изграждане на синтактичното дърво(СД) :

c) низходящ(отгоре -надолу) – когато дървото се строи от корена към листата.
d) възходящ(отдолу нагоре)- когато дървото се строи от листата към корена.

Аргументите на алгоритъма за изграждане на СД са два :

c) програма(низ)
d) граматика

Представянето на програмата е низ от знаци. В този низ знаците за делена на програмата на редове са заменени с разделители(с особени знаци) или пък са описани в граматиката като **разделители**.

Втора стъпка -Трабва да фиксираме представянето на граматиката. Приемаме,че граматиката е от тип LR(1) и налагаме следните съглашениея :

f) На всеки нетерминален символ съпоставяме по единствен начин цяло положително число заградено от <>глови скобки, като на понятието програма съпоставяме <програма> или <0>.
g) Азбука на граматикта наричаме съвкупноста от терминалните и знаци и знаците определени в подточка а).
h) Във всяка металингвистична формула заместваме понятието със съответните им знаци по дефинираната й азбука, като по този начин всяка формула става низ.
i) Сортираме получените низове във възходящ ред.

j) Всяка формула, т.е всеки низ който съдържа повече от една алтернатива заместваме със съответния брой формули без алтернативи, като спазваме последователността на алтернативите.

Полученото множество от низове записваме във таблица с две колонки като понятието отляво от знака по определение записваме в първата колона, а остатъкът от низа без знакът по определение в дясната колона.

Лява колона	Дясна колона
<0>	<……>
<0>	<……>

С низът от з придвижва указателят

Предполага знак в този низ.

Отличителн ъпка разпознава т правила на граматикат някаква синтактичн а чрез целенасоче се реализира по много начи

5. Реализираме логическа функция(предикат) която сръвнява текущият знак от низа на програмата със своя входен аргумент. Нека кръстим **функцията function EUQ(): Boolean**. При открито съвпадение придвижва Pnt напред.
6. На всеки ред от таблицата която описва граматиката, съпоставяме предикат, която връща стойност истина ако разпознае надясно от Pnt подниз породен от правилото записано в дясната колона на този ред. Алгоритъмът на този предикат е линеен и представлява последователност от извиквания на сродни на него предикати и на функцията EUQ. Ако резултатът е лъжа е необходимо указателят Pnt да бъде възстановен със стойността си при влизане в гози предикат. Правилото за образуване на името на такъв предикат е P_<номер на понятие>_<номер на алтернатива>
7. На всяко понятие дефинирано, чрез повече от една алтеранитива, т.е няколко последователни реда от таблицата с еднаква колона се съпоставя предикат с името **P_<номер на понятие>**. Алгоритъмът на този предикат е прост – последователно активиране на предикатите реализиращи алтернативите докато не се поличи резултат истина или не си изчерпат алтернативите. Мерки за възстановяване на първата не са необходими.
8. Ако изпълнение на предиката P_о върне стойност истина или true анализираният низ е правилна програма по отношение на дадената граматика. Този начин на реализация на низходящият граматичен разбор притежава следните достойнства :

d) Разпознавателят на програма се модифицира много лесно ако към граматиката се добавят или отстраняват правила, това означава че ЕП може да бъде развиван лесно.
e) Синтактичното дърво не се строи в явен вид, но е възможно във всеки предикат да се добавят операции за изграждането му като тяхното използване е опционално.

f) Чрез донастройване на така определените предикати лесно се преминава от разпознаване към интерпретация и последваща компилация.

II. Възходящ анализ

Най-късият ляв подниз на програмата, който е непосредствено приводим към синтактична категария(т.е да е в дясната колона на таблицата) се нарича лява основа.Същността на възходящият анализ се заключава в търсене на основа и замяната на тази основа с съответния и нетерминален знак(т.е това е непрекъснато скъсяване на низа отляво) докато се получи нетерминалният знак на аксиомата на граматиката или пък се окаже, че няма основа. Очевидно е, че ефективността на разпознавателя от ефективността зависи от ефективността на алгоритъма за търсене на основи. Една примерна проста реализация на този възходящ анализ се заключава в следното : Изгражда се рекурсивен предикат, чиито глобални аргументи са низа на програмата и таблицата на граматиката. Алгоритъмът му е следният :

d) Търси се последователно низ, определен в дясната колона на таблицата на граматиката , който съвпада с началото на низа на програмата.
e) Ако не бъде открито съвпадение се връща стойност лъжа.
   Ако се открие съвпадение, т-е отново, то основата се замества със знака от лявата колона на таблицата и предикатът се активира отново рекурсивно :
   - Ако рекурсивното активиране върне стойност истина, то тя се връща и като резултат от работата на предиката.
   - Ако рекурсивното извикване върне стойност лъжа, то се възстановява състоянието от преди рекурсивното извикване и се продължава търсенето на основа със следващият низ(ред) от таблицата.

f) Ако активирането на тази процедура върне резултат истина, то програмата е правилна.

След прилагането на някои програмистки техники и хитринки може да се окаже, че възходящият анализ е по-бърз в сравнение с низходящият.Проблемът тук е , че за да се реализира преходът разпознаване, интерпретация , компилация е необходимо да се изградят същите предикати както при низходящият анализ(няма да бъдат предикати, а процедури) и да се натоварят със същите функции(по отношение на превода със изключение на дейността разпознаване). За нашите нужди, ще използваме низходящата разпознаване.

17.Етапи на транслация

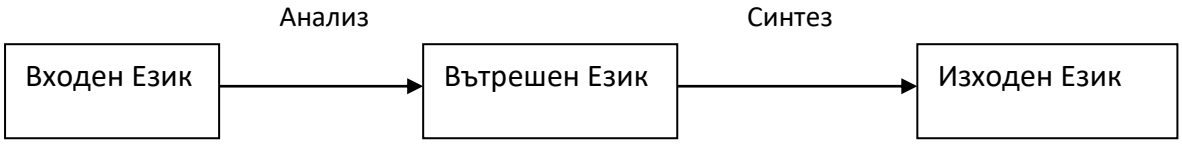
Логически процеса на транслация се състои от два етапа – анализ на входната програма и синтез на изходната програма. От своя страна етапа на анализ съдържа три части, наричани съответентно – лексикален анализ, синтактичен анализ, семантичен анализ. Целта на лексикалния анализ е откриване на лексема от входния виз. По същество той трябва да разпознае всеки подниз, ограден от два разделителя като лексема от някакъв тип. Например име, литерал, служебна дума и т.н. Целта на синтактичния анализ е да открие синтактични структури, програми, подпрограми, като се базира на работата на лексикалния анализатор. Целта на семантичния анализатор е да предаде смисал на разпознатите семантични единици. По същество това е дейност по изграждане на съответни, вътрешни за транслатора, структури от данни, в които се описжат характеристиките на програмните елементи, разпознати от синтактичния анализатор. В алгоритъма му на работа трябва да бъдат закодирани и онези семантични особености на входния език, които не намират отражение в синтаксиса. По същество семантичния анализатор е съвкупност от семантични анализатори, които неявно се приписват на лексикалния и синтактичния анализатор. Примери са кодиране на лексеми, поддържане на таблицата с имена. Може да се каже най-общо, че семантичния анализатор има две основни функции :

а) Отределяне на атрибутите на програмните обекти.
б) Поддържане на необходимите затова, вътрешни за транслатора структури от данни(СД).

Ще предполагаме, че с всеки предикат са свързани три процедури : за лексикален, синтактичен и семантичен анализ. Водеща е процедурата за синтактичен анализ, което означава, че тя активира съответния й лексикален анализатор и в зависимост от резултата на неговата работа използва процедурата за семантичен анализ. Етапа на синтез съдържа две части, наричани : оптимизация и генериране на код. Оптимизацията означава преструктуриране на входната програма с някаква цел, т.е. оптимизатора сам по себе си е някакъв вид транслатор, най-често конвертор. Очевидно е, че тази част не е задължителна за етапа на транслация ,но е много полезна.

Генерирането на код цели произвеждане на изходна програма като се базира на резултатите от етапа на анализ. Това разделяне във времето на работата по анализ и синтез не е задължително. Ако входния език позволява тези етапи могат да се съвместят.

От гледна точка на нашия курс лекции и езика, който ще се превежда ще предполагаме, че с всеки предикат от реализацията на разпознавателя е свързана още една процедура. Ясно е, че тази процедура ще бъде сложна, тъй като в нея ще трябва да се закодира по някакъв начин и лексиката и синтаксиса и семантиката на изходния език.



18.Лексикален анализ

Програма, която извършва лексикален анализ се нарича лексикален анализатор или скенер. Основната задача на всеки такъв скенер е да разпознава лексикалните единици във входния език. Тъй като ще реализираме транслатора, основан на низходящ граматичен разбор ще обсъдим дейността на скенера. Така за наште нужди реализация на скенер означава :

а) Определяна в метаезика на множеството L от видовете понятия на родовото понятие лексема;

б) Разработване на предикатите, съответстващи на тези понятия, като за всеки от тях се предвиди един входен и един изходен параметър;

в) Входният параметър съдържа понятието или терминалия символ, който трябва да се търси във входния низ;

г) Ако търсеното понятие не е открито, то изходния параметър на предиката има неопределена стойност, а самия предикат – стойност „лъжа”.

д) Ако търсеното понятие е открито във входния низ, то изходния параметър съдържа разпознатия подниз и неговото вътрешно представяне. Например ако е разпозната лексема от тип число се връща както самия низ на лексемата, така и двоичния й еквивалент. Ако е разпозната ключова дума се връща както самия низ на ключовата дума, така и нейния вътрешен код за транслатора. Това означава, че всеки предикат от разпознавателя е натоварен и със семантични функции, а именно изграждане на вътрешно представяне за някои или за всички разпознати лексикални единици. Разбира се това конвертиране към вътрешно представяне може да се разположи програмно като семантична функция в предиката, съответстващ на родовото понятие. Но това би била несвойствена за предиката на родовото понятие функция, чието подържане е много трудно в усложията на развиващ се език.

е) Алгоритъмът за търсене бе описан за случая, когато се търси понятие, чиято металингвистична формула съдържа други понятия.

ж) Ако се търси низ от терминални знаци, алгоритъма за търсене е ясен. Търси се съвпадение от позицията на указателя Pnt надясно и при откриване на такова съвпадение указателя Pnt се премества непосредствено след открития подниз като указвания от него знак не трябва да бъде разделител. Това означава, че операцията за придвижване на Pnt трябва разпознава разделителите и коментарите и да ги игнорира.

Необходимо ли е функцията Inc(Pnt) да разпознава ограничителите?

В заключение трябва да отбележим, че ключът към реализирането на ефективен и надежден разпозналател и анализатор съгласно отисаната формална процедура за изграждането им се крие в доброто описание на входния език. От тази гледна точка понятия, които затрудняват изучаването на езика могат да се окажат ключови от гледна точка на реализацията на трнслатор за този език.

19.Синтактичен анализ

Целта на синтактичния анализ е да разпознае синтактичните елементи на програмата, различни от лексеми и ако е необходимо да изгради данновата структура, която ще се изпълни със съдържание от последващия синтактичен анализ.

По същество ние обсъдихме алгоритъма на синтактичния анализатор в Тема 16. Тук ще добавим само някои препоръки от гледна точка на :

А) Избрание от нас подход за граматичен подход;

Б) целта, която имаме е безпроблемно преминаване от разпознавател към интерпретатор и компилатор. За да се постигне тази цел е необходимо:

а) в реализацията да се въведат няколко променливи от логически тип, които указват режима на работа на транслатора. Съвкупността на тези флагове(логически променливи) ще наричаме режим на работа на транслатора. Очевидно е ,че режима на работа може да бъде усложнен чрез детайлизиране на тези флагове. Например : генерация на код, строеж на синтактично дърво и т.н. За да се постигне това се изисква добро предварително проектиране на транслатора и десциплина на реализацията му. Очевидно е, че алгоритъма на всеки предикат трябва да бъде съобразен с режима, в който се използва.

б) за да не строи в явен вид синтактичното дърво е достатъчно да се поддържа системен стек, в който всеки предикат, който реализира алтернативи да записва номера на металингвистичната формула(т.е. името на предиката), чието анализиране е породило неговия успешен край на работа (своеобразно автодокументиране на работата на транслатора);

в) с всеки предикат се свързват една или няколко семантични процедури в зависимост от възможните режими на работа на транслатора и се описва взаимовръзката между тях;

г) тъй като, независимо от режима, множеството от асоциации(таблицата с имената) трябва да се поддържа, е необходимо да се обособо модул от базови семантични процедури.Тяхното предназначение е да реализират операциите позоваване и достъп като използват и поддържат множеството от асоциации.

Обикновено това множество от асоциации се представя чрез т.нар. таблица с имената. По същество тази таблица е стек, чиито редове са асоциации, но при търсене на асоциации тя се разглежда като асоциации, чиито редове се номерират от върха към дъното.

Реализацията на тези четири пункта ще разглеждаме като реализация на синтактичен анализатор. Очевидно става, че реализацията на синтактичния анализатор предполага проектиране на синтактичен анализатор. Освен това трябва да имаме в предвид, че :

А) синтактичния анализатор ползва семантичния амализ, т.е. приключването на работата на семантичния анализатор означава приключване на семантичния анализ.

Б) след края на синтактичния анализ всички имена от програмата са заменени с вътрешни за транслатора имена и тези имена са номер на ред от таблицата с имената.

20.Семантичен анализ

По същество семантичния анализ изпълва със съдържание системните структури от данни на транслатора.

Така както обсъдихме функционалността на базовите синтакти, то за базовите може да се каже, че тяхната функционалност се заключва в генериране на позиции на техните програмни елементи. Добре е техните дескриптори да описват всички характеристики на разпознатите програмни елементи независимо от това, че :

А) някои от характеристиките могат да бъдат наследени от други програмни елементи или пък да са функция на други елементи;

Б) някои от характеристиките могат да не бъдат използвани в етапа на синтез;

Този подход ще позволи да се реализират множество варианти на работа на транслатора в етапа синтез. Съвкупността от всички даннови структури, описващи една програма трябва да се разглеждат като позиция на програма. Тази гледна точка решава проблема с асоциациите на подпрограма. Разбира се в този случай операцията търсене на асоциация би се усложнила в случай на статична област на действие. Този проблем има елегантно решение в следните няколко пункта :

а) Поддържа се единна таблица с имена, в която имената се записват е реда на срещането им в програмата, съгласно правилото за четене;

б) на всяка подпрограма съответства единствена асоциация в тази таблица;

в) кодът на позиция на всяка подпрограма е множество(списък) от кодове предопределени от възможните режими на синтез. Например : за интерпретация кодът трябва да съдържа текста на самата подпрограма, а при компилация – текста на преведената програма.

Извода е, че реализацията на семантичния анализатор предполага проектиране на етапа синтез на програма.

21.Интерпреация на програма

По същество интерпретацията представлява емулация на входния език в термините на реализационния език. От гледна точка на избрания от нас подход това означава на всеки предикат да се съпоставя интерпретационна процедура или метод, която се изпълнява в режим на интерпретация преди изходда от този предикат със стойност истина. Алгоритмите на тези интерпретационни процедури са известни от семантиката на входния език. Остава единствено за реализацията на тези алгоритми да бъдат подбрани или избрани онези най-прости изразни средства на реализационния език, които :

А) които имат директен аналог в термините на изходния език;

Б) известен е преводът им в изходния език. Очевидно най-добре е да се използва семантиката на изходния език, като се пише в термините на реализационния език. Това е най-правилният подход, защото се проявяват онези вътрешни обекти и операции, които в последствие ще трябва да се закодират в термините на изходния език.

В) преминаването от интерпретация в генерация на код става рутинна дейност;

Общия извод е, че реализацията на интерпретатор изисква много добро владеене на семантиката на реализационния език от гледна точка на семантиката на изходния език. В частност необходима е емулация на механизма з управление на паметта на БВИМ, т.е. на изходния език, като реализацията на интерпретатора трябва да започне с реализацията на този механизъм.

22.Генерация на код

По същество генерацията на код е най-неангажиращата дейност в процеса на разработване на транслатор. Генерацията на код се разполага в :

А) прост превод на алгоритмите на интерпретационните процедури в термините на изходния език;

Б) Оформление на новополучения текст в съответствие със синтаксиса на изходния език.

От гледна точка на избрания от нас подход подусловието Б) би трябвало да отпадне. Разбира се при прехода би трябвало да се отчете, че част от текста на интерпретационните процедури е излишен. Например : текстове, които описват механизмите за управление (данни, паметта, послед. действия) на БВИМ трябва да отпаднат. От гледна точка на генерацията на код тези текстове само генерират еквивалентност на входната и генерираната изходна програма.

Изводът е, че най-тежък като работа е интерпретаторът, т.к. при неговата реализация разработчикът на транслатор трябва да използва едновременно три семантики и поне два синтаксиса на входния иреализационния език.

23.Оптимизация

Под оптимизация на програма се разбира нейната обработка с цел да бъде получена по-ефективна обектна програма. За съжаление досега не е открит универсален алгоритъм за получаване на абсолютно ефективна програма. Поради това терминът оптимизация ще разбираме в смисъл подобряване на някакво качество на програмата при условие на запазване на другите й качества в някакви граници. Различаваме две групи оптимизации – машинно независима и машинно зависима оптимизация.

1. Машинно-независима оптимизация – тази оптимизация се извършва върху вътрешното представяне на входната програма. Тя е съвкупност от методи, целящи:
   А) Изчиаляване на изрази формирани от литерали;
   Б) Елиминиране на повтарящи се изчисления;
   В) Отстраняване на безсмислени оператори или операции;
   Г) Минимизиране на броя на използваните променливи;

Д) Минимизиране на броя на операциите позоваване и достъп(особено в цикли);
Е) Минимизиране броя на формалните парамтри на подпрограмите;
Ж) Прилагане на правилото на копиране за еднократно активирани подпрограми;
З) Прилагане на правилото за копиране за онези подпрограми, чийто код е минимален в сравнение с кода на алгоритъма за предаване на параметри;
И) Изнасяне извън цикъл на операции, чиито аргументи не зависят от параметъра на цикъла;
Й) Замяна на операции в цикъл с по-бързи такива;
К) Линеализация на многомерни масиви в цикъл;
Л) Обединяване, сливане и разчленяване на цикли;
М) и т.н.

2. Машинно-зависима оптимизация

Тя се извършва върху вътрешното представяне на изходната програма(ако се поддържа такова). Тази оптимизация зависи от конкретната БВИМ и обикновено цели използването на най-бързите команди и рационалното разпределение на различните видове памет. Основните алгоритми, които се използват при този вид оптимизация са :

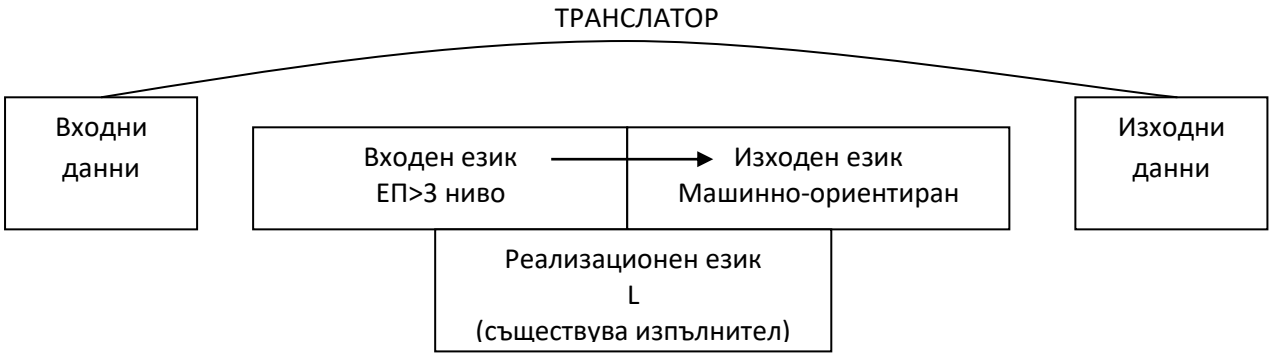
А) Минимизация на броя на използваните регистри;
Б) Минимизация на броя на прехвърлянето на данни между регистрите и ОПамет и обратно;
В) Ефективно използване на индексните регистри при цикли и индексни изрази;

Обикновено оптимизатора се реализира в отделен модул на програмата, тъй като алгоритмите му са сложни и тежки и не е необходимо да бъдат прилагани при всяка транслация.

В кои случай не е необходимо да се оптимизира входната програма?

24.Схеми за реализация на транслатор

Всеки разработчик на транслатор използва три езика – входен, реализационен и изходен език. Тази триада от езици схематично се изобразява чрез така наречените Т-диаграми.

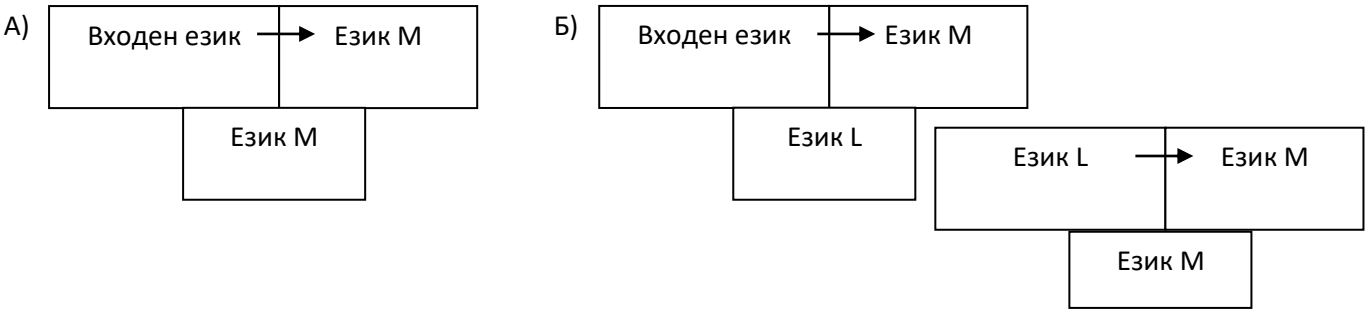


Очевидно съвпадението на входния и реализационния език е невъзможно, а в общия случай съвпадение на входния и изходния език е безсмислено. Следователно наредената двойка

(реализационене език, изходен език) определя основните схеми за реализация на транслатора. Обикновено терминът транслатор се употребява :

А) Или когато РЕ и ИЕ съвпадат;
Б) Или когато под РЕ съществува транслатор до ИЕ;

Казаното се илюстрира чрез следнире схеми:

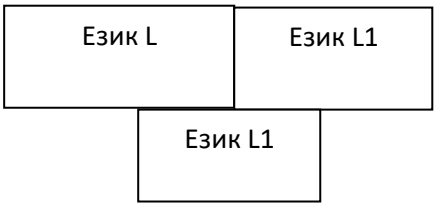


Втората диаграма показва, че реализационния език може да се счита за машинно-ориентиран език.

Ако изходния език е език от ВН, но за него ина транслатор, полуяваме :

Тази схема показва, че изходния език също може да се счита за машинно ориентиран език. Следователно термина транслатор употребяваме, когато реализационнитят и изходния език са машинно ориентирани и съвпадат. Когато РЕ и ИЕ са машиноо-ориентирани, но не съвпадат се употребява термина прост транслатор. Това означава, че изходната програма ще реботи на ВИМ, различна от тази на която работи транслаторът.

Очвидно реализацията на транслатор в термините на машинно-ориентиран реализационен език е тежка работа. По-добре е реализационния език да бъде език от високо ниво, но тогава е възможно реализационния език да бъде подмножество на входния език, т.е. казва се, че входния език е диалект на реализационния език.



L1 е подмножество на L.

Тази идея е позната като възможност за нарастване на изразителната сила на реализационния език със собствени сили.

Съществуват различни схеми за реализация на транслатори с този механизъм като целта е минимизация на времето и усилията по реализация на транслатора.