Формални Езици и Езикови Процесори
ТУ, кат. КС, летен семестър 2012

Лекция 7

Тема:

Компилатори
(общ преглед, обзор)

Съдържание:

- Основна постановка.
- Структура на компилатор.
- Контекст на компилатор.
- Модели на компилатор (Tremblay/Sorenson, Aho/Sethi/Ullman, Holub, Янков).
- Примери.

Езикови Процесори

1. Езикови Процесори
   - Транслатори, Асемблери, Конвертори
1. Транслатори - класификация
   - Компилатори
   - Интерпретатори

Компилатори

A **compiler** is a computer program (or set of programs) that translates text written in a computer language (the *source language*) into another computer language (the *target language*). The original sequence is usually called the *source code* and the output called *object code*. Commonly the output has a form suitable for processing by other programs (e.g., a linker).

The most common reason for wanting to translate source code is to create an executable program.

Компилатори

The term "compiler" is used for programs that translate source code from a HLL to a low LL (e.g., assembly language or machine language).

A program that translates from a low level language to a higher level one is a *decompiler*.

A program that translates between high-level languages is usually called a *language translator*, *source2source translator*, or *language converter*.

A *language rewriter* is usually a program that translates the form of expressions without a change of language.

Резидентни с/у Крос К.

A compiler may produce binary output intended to run on the same type of computer and operating system ("platform ") as the compiler itself runs on. This is called a native-code compiler.

Alternatively, it might produce binary output designed to run on a different platform. This is known as a cross compiler.

Справка: Т-диаграми от лекция 1.

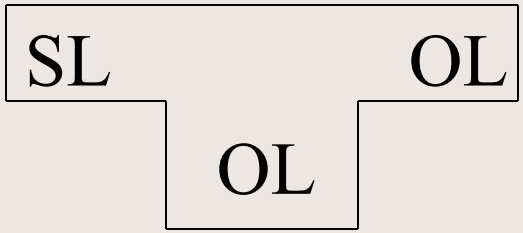
Само Копилируеми К.

A compiler may read source code written in the same language as the compiler implementation language. This is called a steady state compiler or self-compiling compile
.

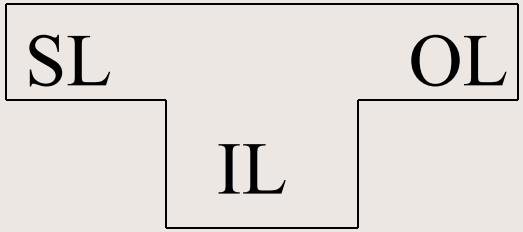
Справка: Т-диаграми от лекция 1.

Т - Диаграми

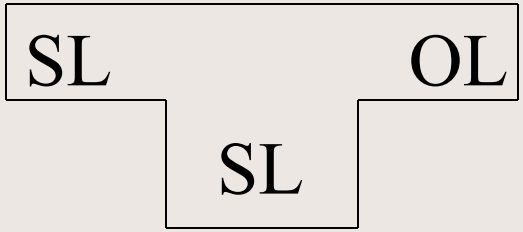
Резидентен Компилатор



Крос Компилатор



Само компилируем К.



Компилатори

Типични обработки, провеждани от К.:

Preprocessing

lexing,

parsing,

semantic analysis,

code optimizations,

code generation.

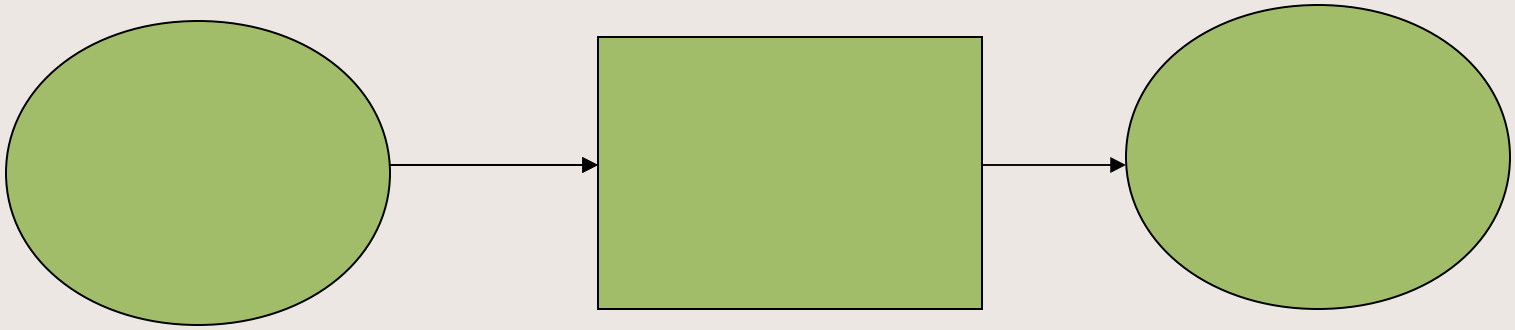
Припомняне
на
теми от уводна лекция 1

ЕП са мета програми

Първична програма

Езиков процесор

Обектна програма



Видове транслатори

- Компилатори
- Интерпретатори

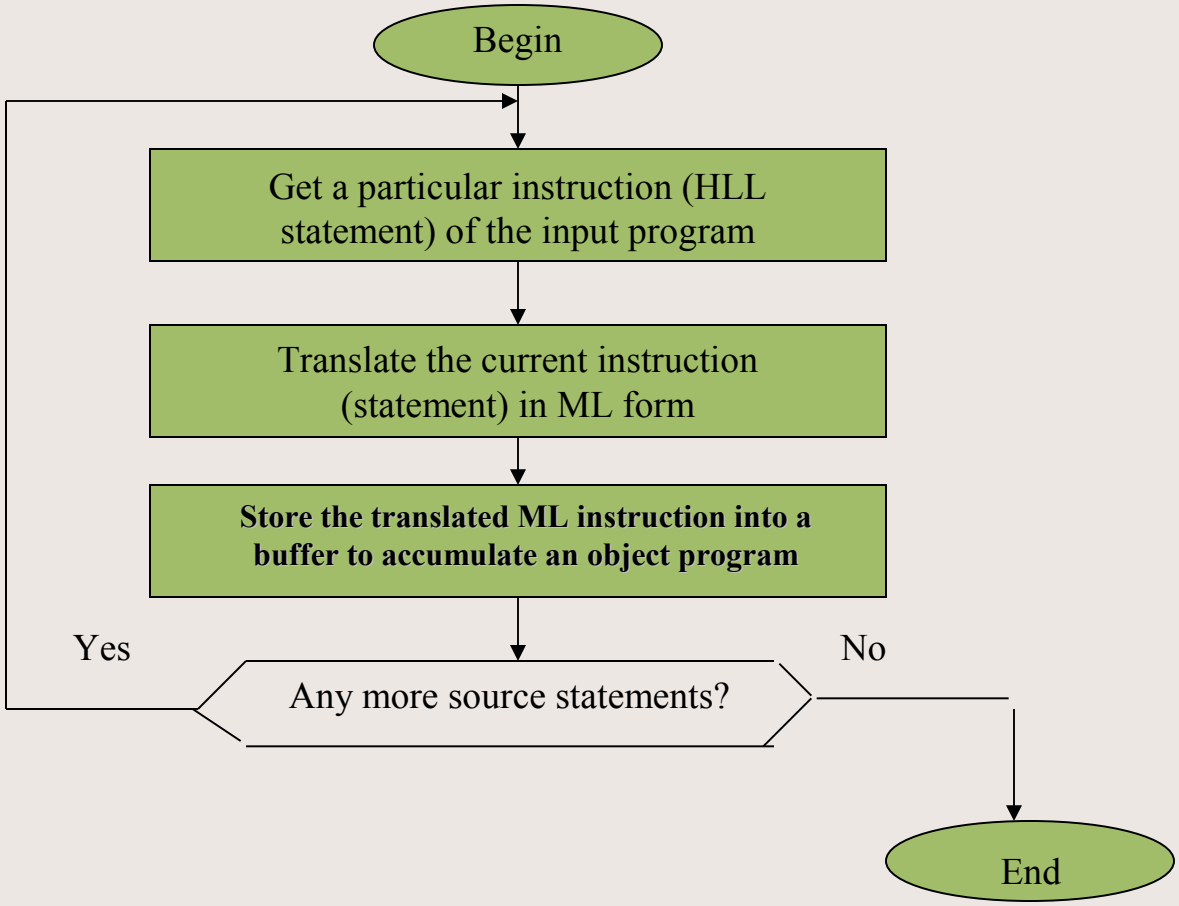
Алгоритъм на К. и И.

- Блок схема на обобщен алгоритъм на компилатор
- Блок схема на обобщен алгоритъм на интерпретатор

Обобщен алгоритъм на компилатор

1. Начало.
2. Четене на EBH инструкция (HLL statement) от вх. Програма.
3. Превод на текущата инструкция в ME формат.
4. **Буфериране на превода за формиране на обектна програма.**
5. Има ли още инструкции на входа? Ако Да, премини към 2. Иначе, премини към 6.
6. Край.

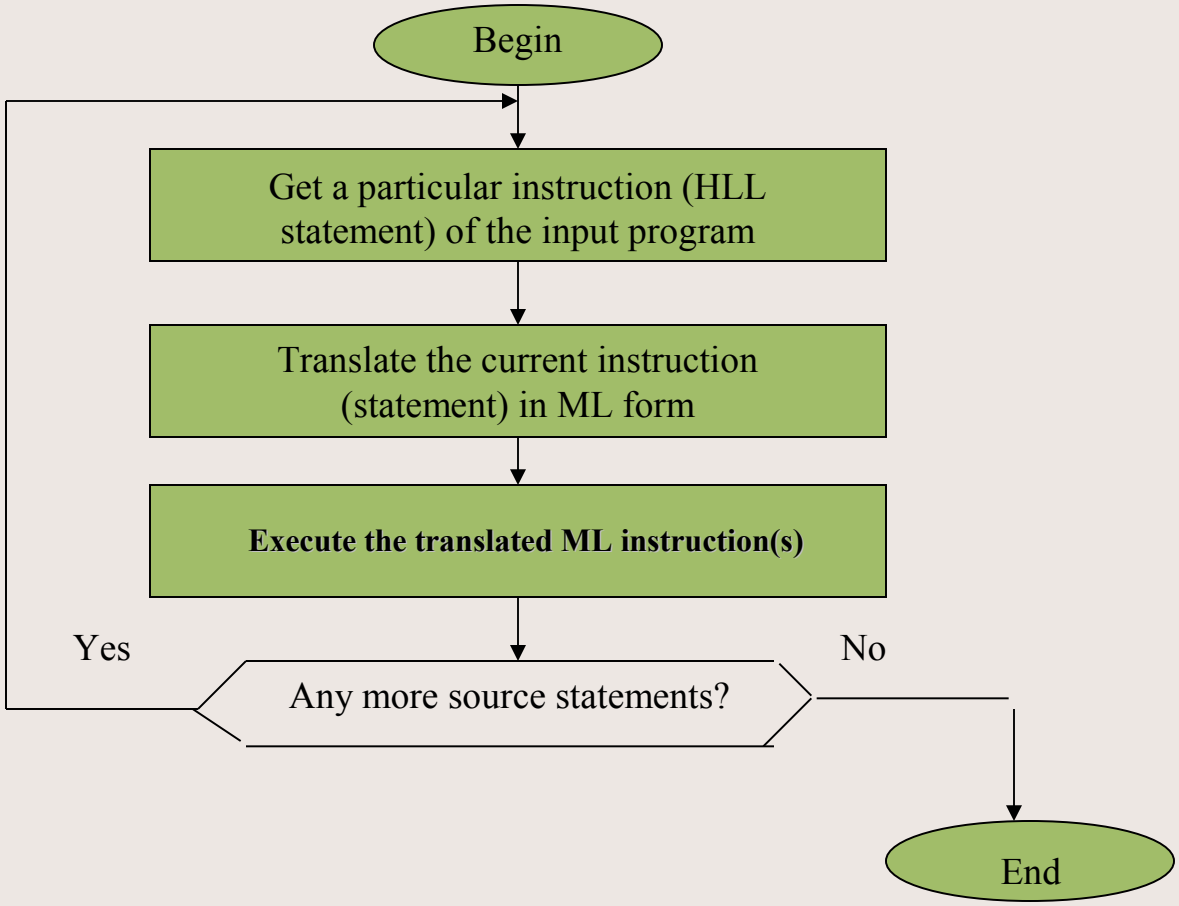
Обобщен алгоритъм на компилатор



Обобщен алгоритъм на интерпретатор

1. Начало.
2. Четене на EBH инструкция (HLL statement) от вх. Програма.
3. Превод на текущата инструкция в ME формат.
4. **Директно изпълнение на превода.**
5. Има ли още инструкции на входа? Ако Да, премини към 2. Иначе, премини към 6.
6. Край.

Обобщен алгоритъм на интерпретатор

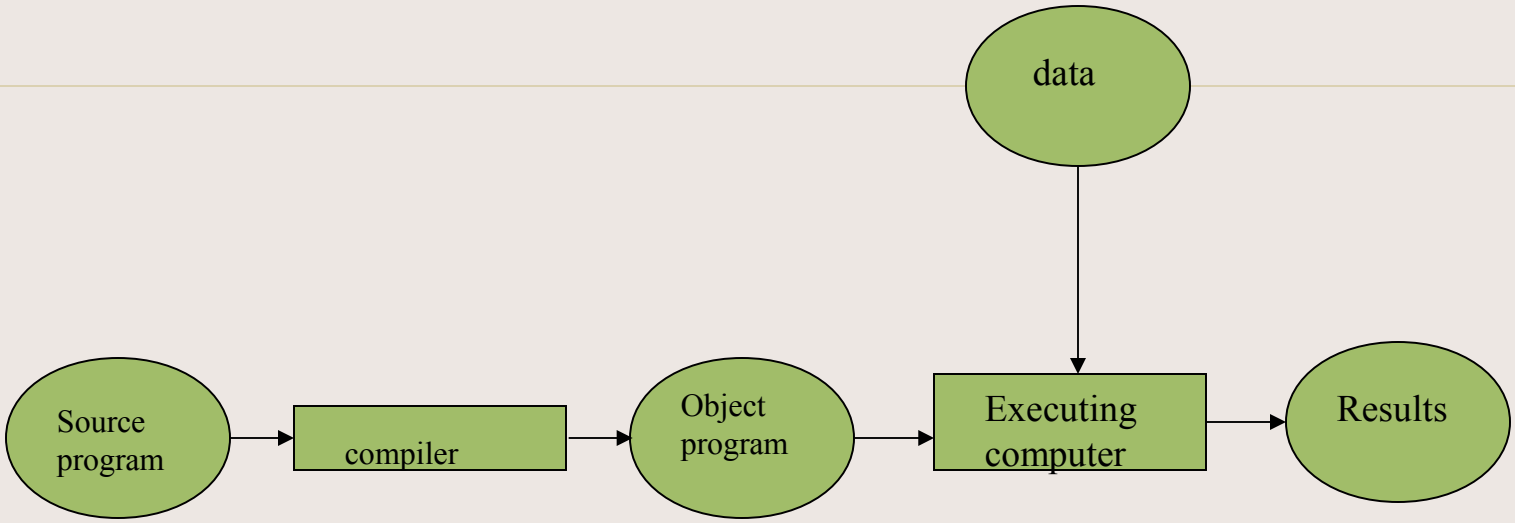


Сравнение К. - И.

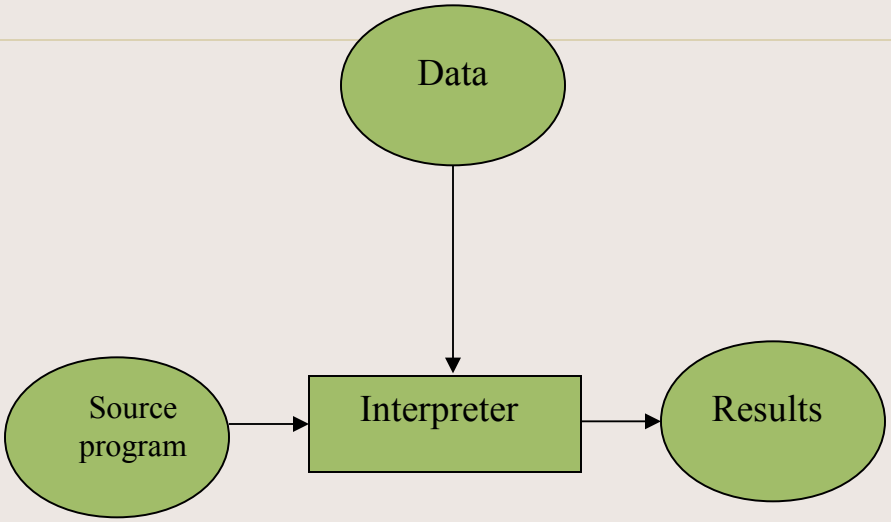
- И. се пише по-лесно от К.
- Изпълнението на една компилирана програма е по-бързо от изпълнението на една интерпретирана програма.
- К: Фаза компилация и изпълнение са отделни.
- И: Фаза компилация и изпълнение са съвместени.
- К: Веднъж компилирана, програма може многократно да се изпълнява.
- И: Изпълнение винаги се съпътства с превод (транслация).

Сравнение К. - И.

- Илюстрация на процес компилация – времето за компилация и времето за изпълнение са разделени;
- Илюстрация на процес интерпретация – времето за компилация и времето за изпълнение са съвместени.



Фаза компилация Фаза изпълнение



Фаза интерпретация

Контекст на компилатор

В допълнение на К., няколко други обработки се извършват.

Preprocessing – първичната програма е структурирана в модули, като отделни файлове.

Обектната програма на изхода на компилатора се нуждае от следващо процесиране – превод на асемблер в преместваем код, свързване на обектен код с ПП от статични библиотеки и др.

Контекст на компилатор

Първична програма на макроезик

Препроцесор

Първична програма

Компилатор

Превод в код на АсемблеренЕзик

Асемблер

Преместваем машинен код /Relocatable code/

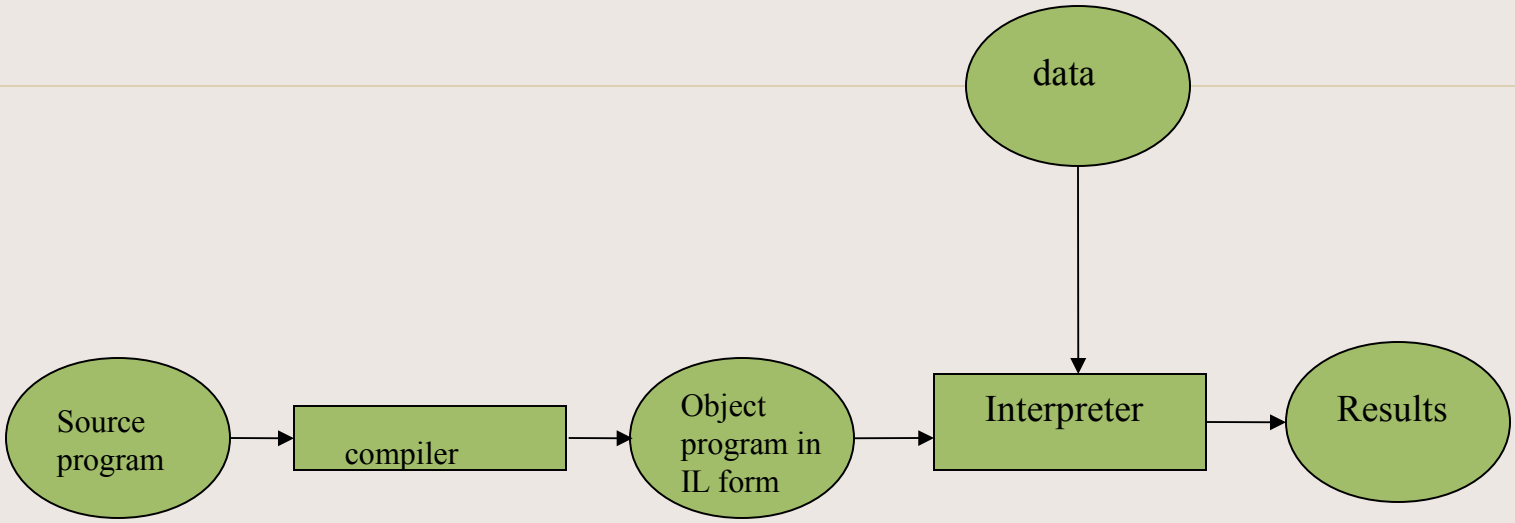
Линкер/Linker <библ.об.модули>

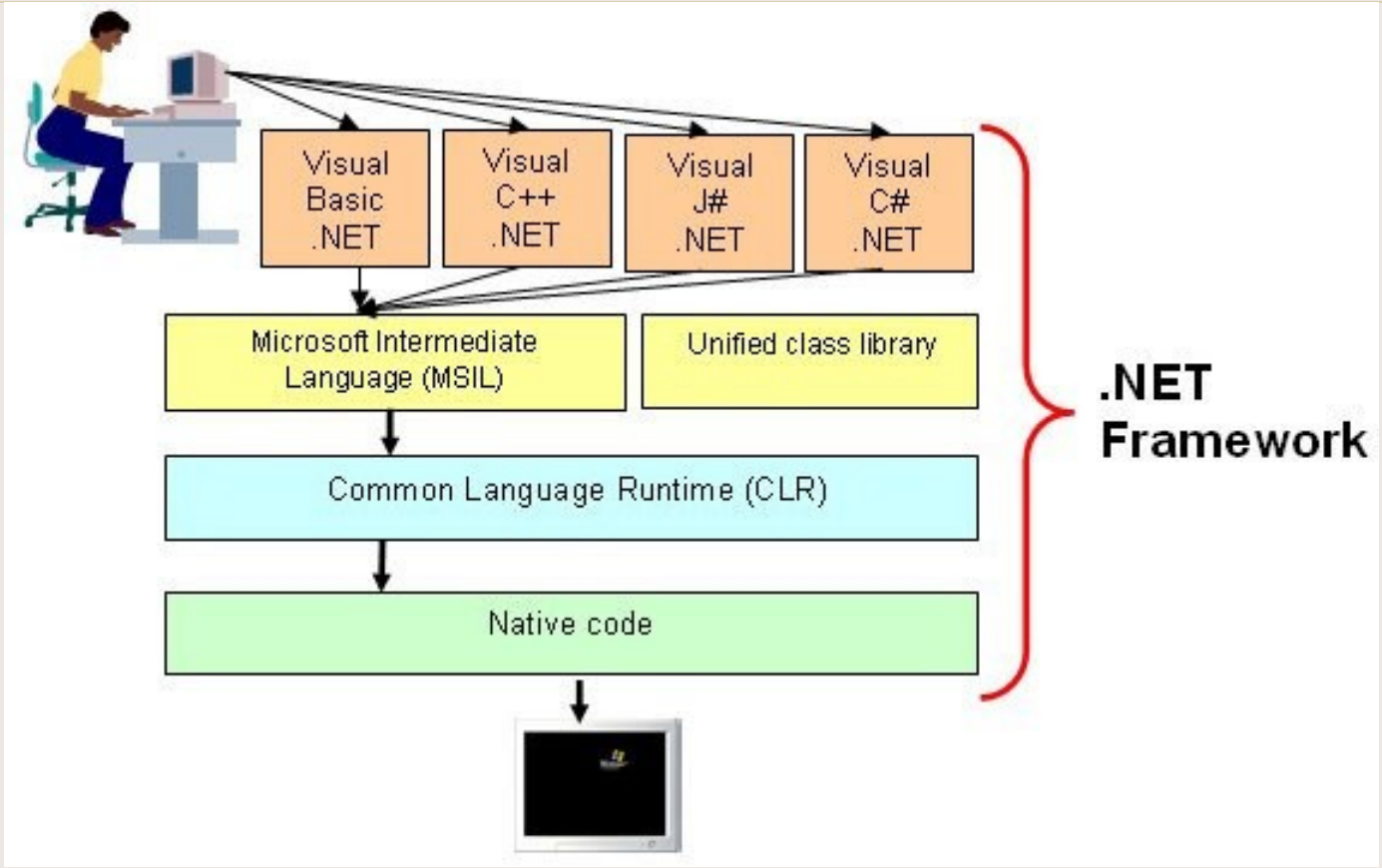
Изпълнима програма МЕ

Междинни Езици

Смесена /хибридна/ схема на ЕП:
Компилация от полу-интерпретиращ тип

Приложения с междинен код –
примери на PASCAL >> P-code
MBPL >> MK
JAVA >> byte code
Microsoft .NET >> MSIL code


data
Source program
compiler
Object program in IL form
Interpreter
Results

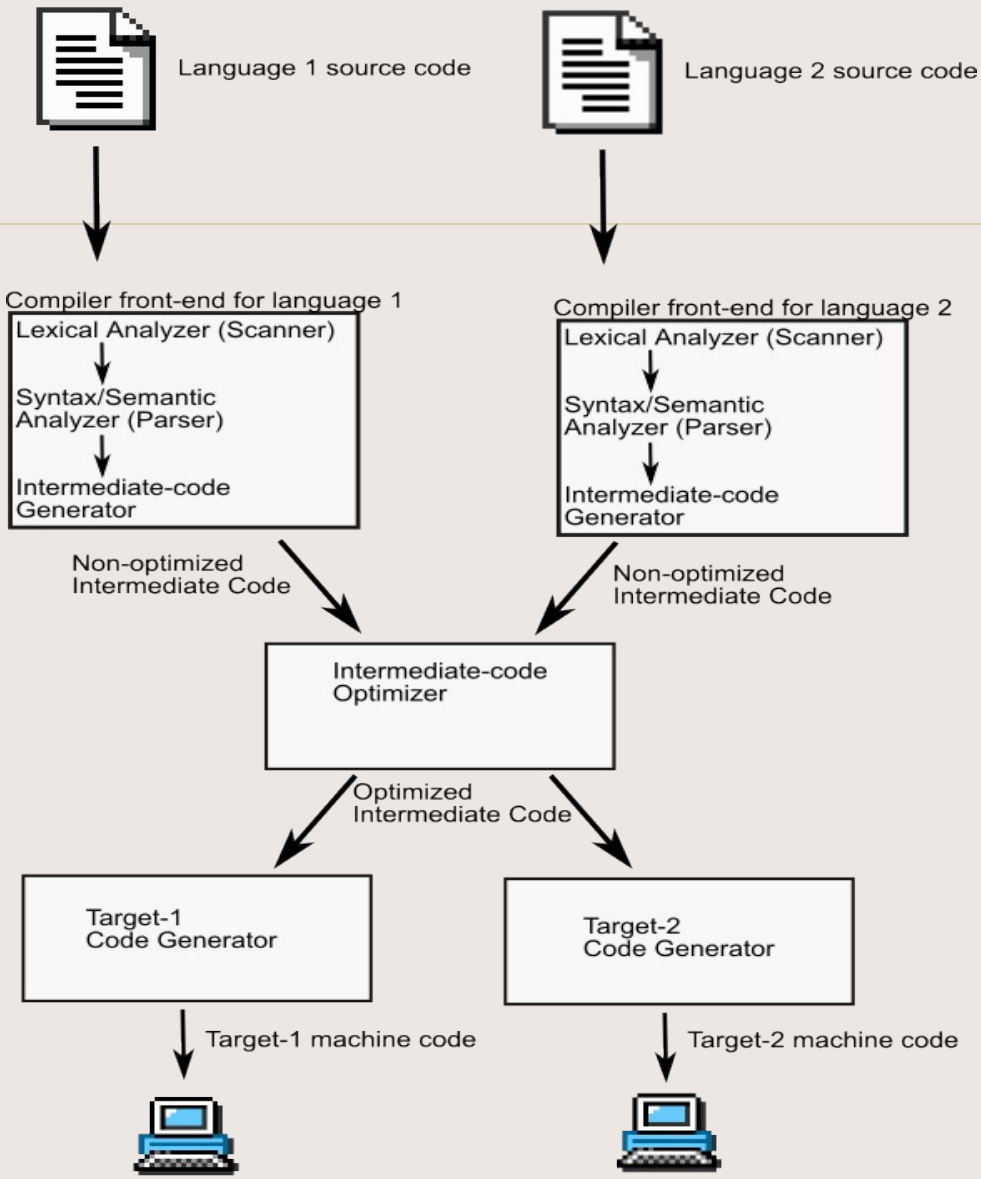

Visual Basic .NET
Visual C++ .NET
Visual J# .NET
Visual C# .NET
Microsoft Intermediate Language (MSIL)
Unified class library
Common Language Runtime (CLR)
Native code
.NET Framework

Обработки в .NET

- The VBasic, C++, J#, C# compiler job is to turn the source code into a working program.
- .NET работи с междинен език MSIL (MicroSoft Intermediate Language (или съкратено IL).
- The compiler reads your source code and produces IL code.
- The .NET JIT compiler then reads your IL code and produces an executable application in memory.

Схема на работа
на
много-езиков /multi-language/,
много-целеви /multi-target/
компилатор

е показана на следния слайд


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized Intermediate Code
Non-optimized Intermediate Code
Intermediate-code Optimizer
Optimized Intermediate Code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

Модел(и) на Компилатор

Термини – фаза(етап) и пас

Фаза(етап) на компилация е независима задача в процеса на компилация (independent task used in compilation process).

Пас на компилация – процес на цялостно обхождане на първичния програмен текст в процеса на компилация (едно-пасови и много-пасови компилатори).

Модел(и) на Компилатор

Лексически анализ,

Синтактичен анализ,

Семантичен анализ,

Генерация на код,

Оптимизация са фази, независими задачи.

Известни са реализации на едно-пасови компилатори.

Известни са реализации на компилатори с 9, 11, дори повече от 30 паса.

Модел(и) на Компилатор

Фактори, влияещи на броя пасове:

Наличната RAM памет;

Бързодействието и размерът на компилатора;

Бързодействието и размерът на об. програма;

Вграждане на средства за настройка;

Стратегия за откриване и отстраняване на грешки;

Числеността на колектива разработващ К.

Модел(и) на Компилатор

– Боян Янков (TOC);

Лексически анализ	Lexical Analyzer
Синтактичен анализ	Syntax Analyzer
Семантичен анализ и генерация на код	Semantic Analyzer and Code Generator
Оптимизатор на кода	Code Optimizer
Таблици	Tables

Модел(и) на Компилатор

– A.Holub (Compiler Design in C);

Preprocessor	pass 1	препроцесор
Lexical analyzer	pass 2	лекс. А-тор
Symbol tables		симв. таблици
Syntactic analyzer		синт. А-тор
Code generator		код генератор
Optimization	pass 3	оптимизатор
Back end	pass 4	закл. обработка

Модел(и) на Компилатор

– J.P.Tremblay, P.Sorenson (Theory and Practice of Compiler Writing);

Analysis	Анализ
lexical analyzer	лекс. анализатор
syntactic analyzer	синт. анализатор
semantic analyzer	сем. анализатор
Synthesis	Синтез
code generator	код генератор
code optimizer	код оптимизатор
Tables	

Модел(и) на Компилатор

– A.Aho, R.Sethi, J.Ullman (Compilers – Principles, Techniques and Tools).

Лексически Анализатор;

Синтактичен Анализатор;

Семантичен анализатор

Генератор на междинен код

Оптимизатор на кода

Генератор на об. код

Обработка на таблици Symbol table manager

Обработка на грешки Error handler

Модел(и) на Компилатор

– A.Aho, M.Lam, R.Sethi, J.Ullman (Compilers – Principles, Techniques &Tools)

Лексически Анализатор;

Синтактичен Анализатор;

Семантичен анализатор

Генератор на междинен код

Машинно-независим Оптимизатор на кода

Генератор на об. код

Машинно-зависим Оптимизатор на кода

Обработка на символни таблици

Заключение

Превод на първична програма в обектна програма (фази, етапи, пасове):

- Лексически Анализ;
- Синтактичен Анализ;
- Управление на таблици;
- Откриване на грешки и възстановяване;
- Семантичен Анализ;
- Генерация на междинен код;
- Оптимизация на кода;
- Генерация на обектна програма.

Лексически Анализ

Лексически Анализ

- Първичната програма е низ от символи – букви, цифри, разделители.
- Първичната програма съдържа елементарни езикови конструкти – променливи, константи, запазени думи, операции, разделители. Очаква се компилаторът да разпознае тези конструкти и идентифицира като лексеми.
- ЛА чете входната програма и генерира поредица от описатели на лексеми, наречени токени /tokens/. Пример

```
test: if a > b then x=y;
```

test: if a > b then x = y;

• Идентификатор - етикет	test
• Разделител	:
• Запазена дума	if
• Потреб. Деф. име	a
• Разделител rel операция	>
• Потреб. Деф. име	b
• Запазена дума	then
• Потреб. Деф. име	x
• Разделител	=
• Потреб. Деф. име	y
• Разделител	;

Лексически Анализ

- Процесът се нарича scanning.
- Обработващата програма се нарича scanner.

Синтактичен Анализ

Синтактичен Анализ

- Цел: групиране на токените в по-общи категории като изрази, оператори, процедури и дори програми.
- Изход: дърво на разбора или негов еквивалент.
- (A+ B) * (C + D)

Синтактичен Анализ

- Дърво на разбора се строи според продукциите на граматиката КСГ.
- Процесът се нарича parsing.
- Обработващата фаза се нарича parser.

Семантичен Анализ

Семантичен Анализ

- Цел: да се определи смисълът /семантиката/ на първичната програма.
- (A + B) * (C + D)
- Дефинирани ли са операндите?
- Дефинирани ли са операндите > 1 път ?
- Имат ли операндите един и същ тип?
- Имат ли стойност операндите?

Генерация на Междинен Код

Генерация на междинен код

- IL/МК/ може да се представи по много начини. Една възможна форма е:
- Представяне в тетради/четворки

  (CodeOp, Oprnd1, Oprnd2, Result)

- Формата с тетради е прието да се нарича ТАС три адресен код /**three address code/.**

Три адресен код /three address code/

Всеки запис TAC се представя като тетрада quadruple (operator, operand1, operand2, result).

Елементарен оператор /statement/ има общ вид

$$x := y \; op \; z$$

Където x, y и z са променливи, константи или временни променливи, генерирани от К., а *op* представлява коя да е двуместна операция.

$$(op, y, z, x)$$

Три адресен код /three address code/

Изрази с повече от една операция като:

$$p := x + y * z$$

Не могат да се представят с един запис ТАС. Такъв тип изрази се декомпозират в поредица от ТАС записи, например,

$t1 := y * z$	$(*, y, z, t1)$
$p := x + t1$	$(+, x, t1, p)$

Три адресен код /three address code/

TAC записи се прилагат и при изрази с едноместни операции.

Пример:

$x := op\ y$ $\qquad\qquad$ $(\ op,\ y,\ -,\ x\ )$

Генерация на междинен код

- Абстрактна форма тетради

  (CodeOp, Oprnd1, Oprnd2, Result)

(A+B)*(C+D) се превежда в 3 тетради

(+, A, B, T1)

(+, C, D, T2)

(*, T1, T2, T3)

Генератор на код

```
//          ( +, A, B, T1 )
                    LDA  A
                    ADD  B
                    STA   T1
//          ( +, C, D, T2 )
                    LDA  C
                    ADD  D
                    STA   T2
//          ( *, T1, T2, T3 )
                    LDA  T1
                    MUL  T2
                    STA   T3
```

Оптимизатор на Код

Оптимизатор на код

// (+, A, B, T1)		
LDA A	LDA A	LDA A
ADD B	ADD B	ADD B
STA T1	STA T1	STA T1
// (+, C, D, T2)		
LDA C	LDA C	LDA C
ADD D	ADD D	ADD D
STA T2	***STA T2***	
// (*, T1, T2, T3)		
LDA T1	***LDA T2***	
MUL T2	MUL T1	MUL T1
STA T3	STA T3	STA T2

Пълноценен пример

Пълноценен пример

Ръчна компилация на един C/C++/C#/Java първичен оператор за присвояване:

```
pos = init + rate * 60;
```

- Три потребителски дефинирани имена идентификатори, и трите с реален тип (`double pos, init, rate;`)
- Една константа – цяла десетична, `60`

pos = init + rate * 60;

- Лексически анализ
- Таблици–identifiers(pos,init,rate),constants(60)
- Синтактичен анализ – parse tree
- Синтактичен анализ – syntax tree
- Семантичен анализ – модиф. syntax tree
- Обработка на грешки
- Генерация на междинен код
- Оптимизация на код
- Генерация на код

Лексически анализ

Лексически анализ или линеен анализ, или Scanning сканира входната програма Л >> Д и групира символите в токени/дескриптори/ с колективно/групово/ значение.

```
pos = init + rate * 60;
```

Лексически анализ

Допълнителни дейности:

```
pos = init + rate * 60;

pos  =   init  +  rate  *     60;
```

- Интервалите се игнорират.
- Обработват се много-редови оператори.
- ЛА запълва динамичните таблици на имена и константи.

Лексически анализ

```
pos = init + rate * 60;
```

Lexical analyzer (scanner)

<id1> = <id2> + <id3> * <intconst>;

Таблици

pos = init + rate * 60;

Таблица на имената

<id1> ***pos*** атрибути: type real, size 1, scalar var, etc

<id2> ***init*** атрибути: type real, size 1, scalar var, etc

<id3> ***rate*** атрибути: type real, size 1, scalar var, etc

Таблица на константите

<intconst> ***60*** атрибути: type integer, value 60, etc

Синтактичен Анализ

Дърво на разбора в две форми:

Parse tree – създадено според граматиката.

Syntax tree – компресирано представяне на parse tree, в което операциите са възли на графа, а операндите са дъщерни възли на възела операция.

Синтактичен анализ

Parse tree

Илюстрация по следната КСГ граматика:

<assgn stmt> ::= <id> = <E> ;

<id> ::= let | <id> let | <id> dig

<E> ::= <T> | <E> + <T>

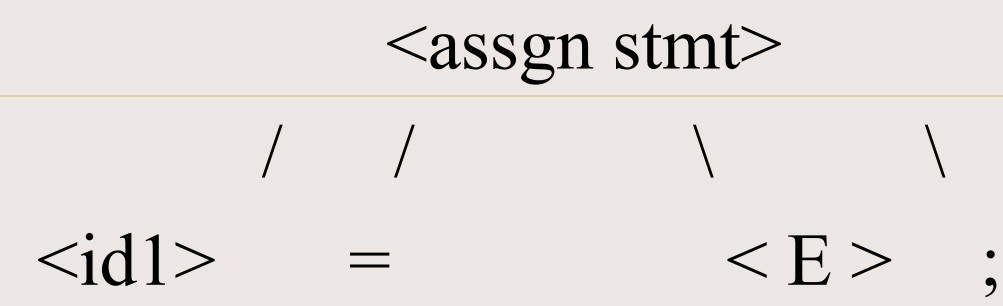
<T> ::= <F> | <T> * <F>

<F> ::= <id> | (<E>)

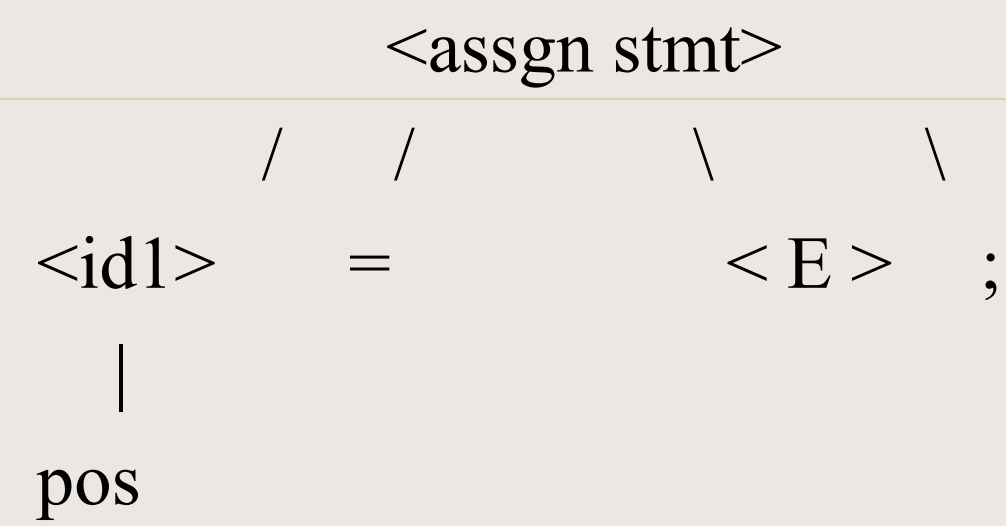
Parse tree

<assgn stmt> - корен на дърво

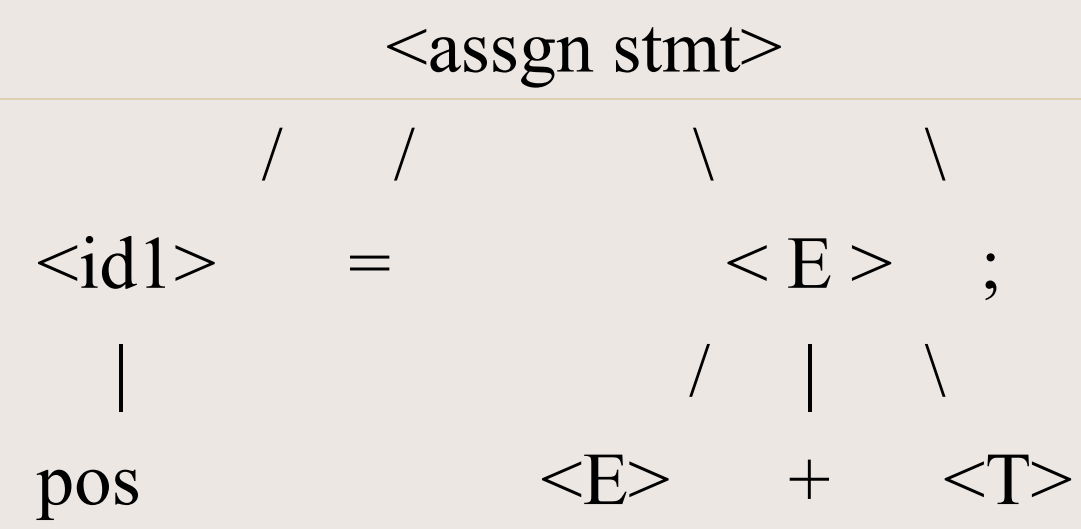
Parse tree



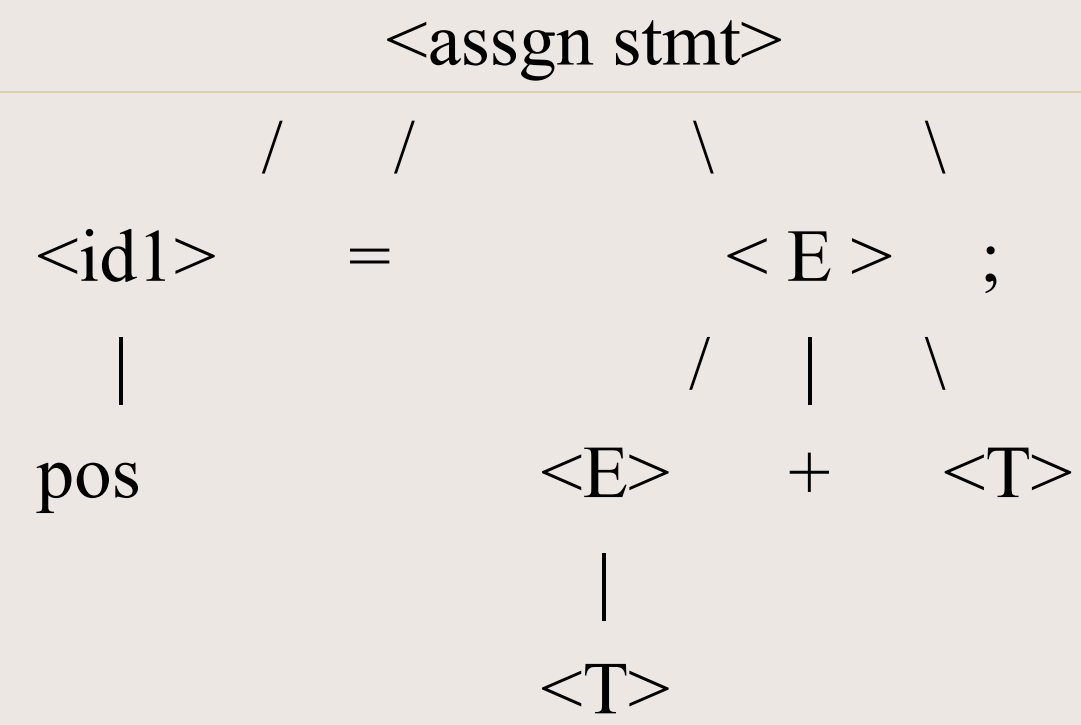
Parse tree



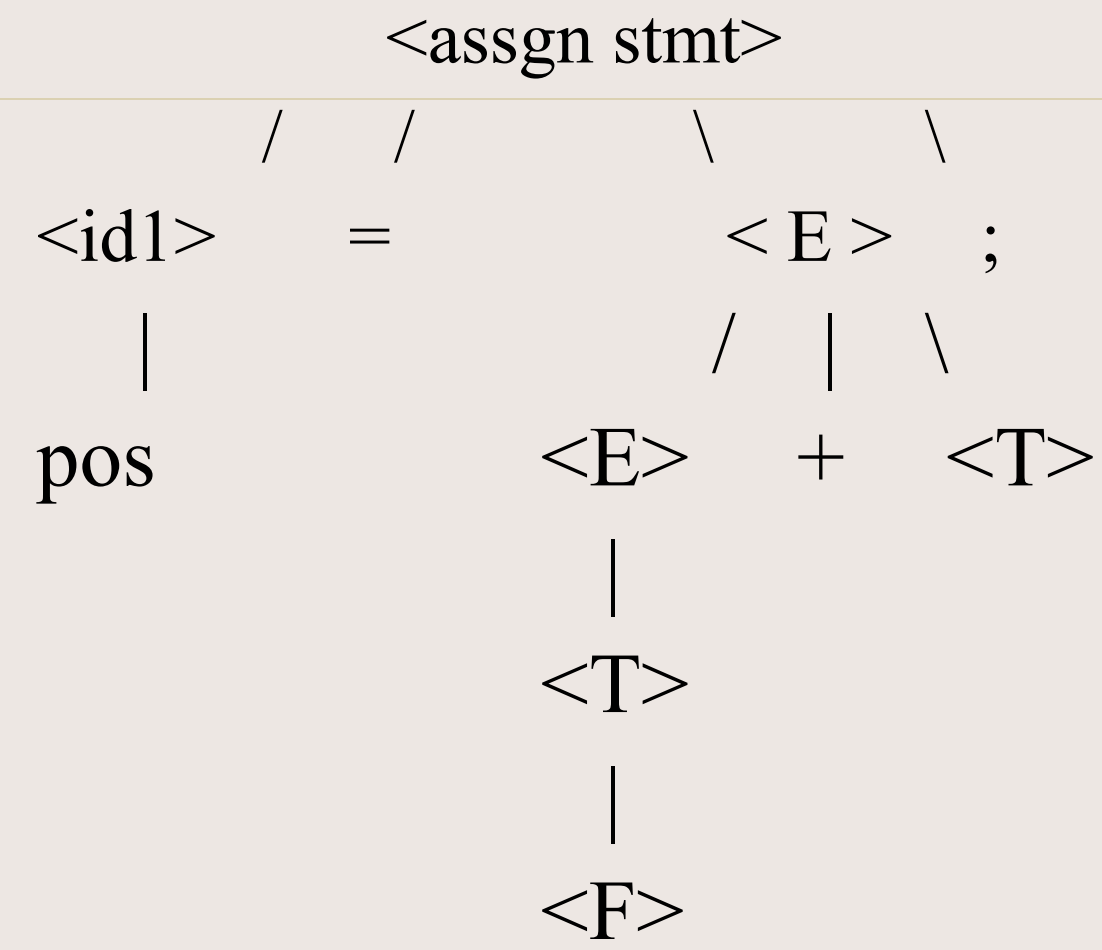
Parse tree



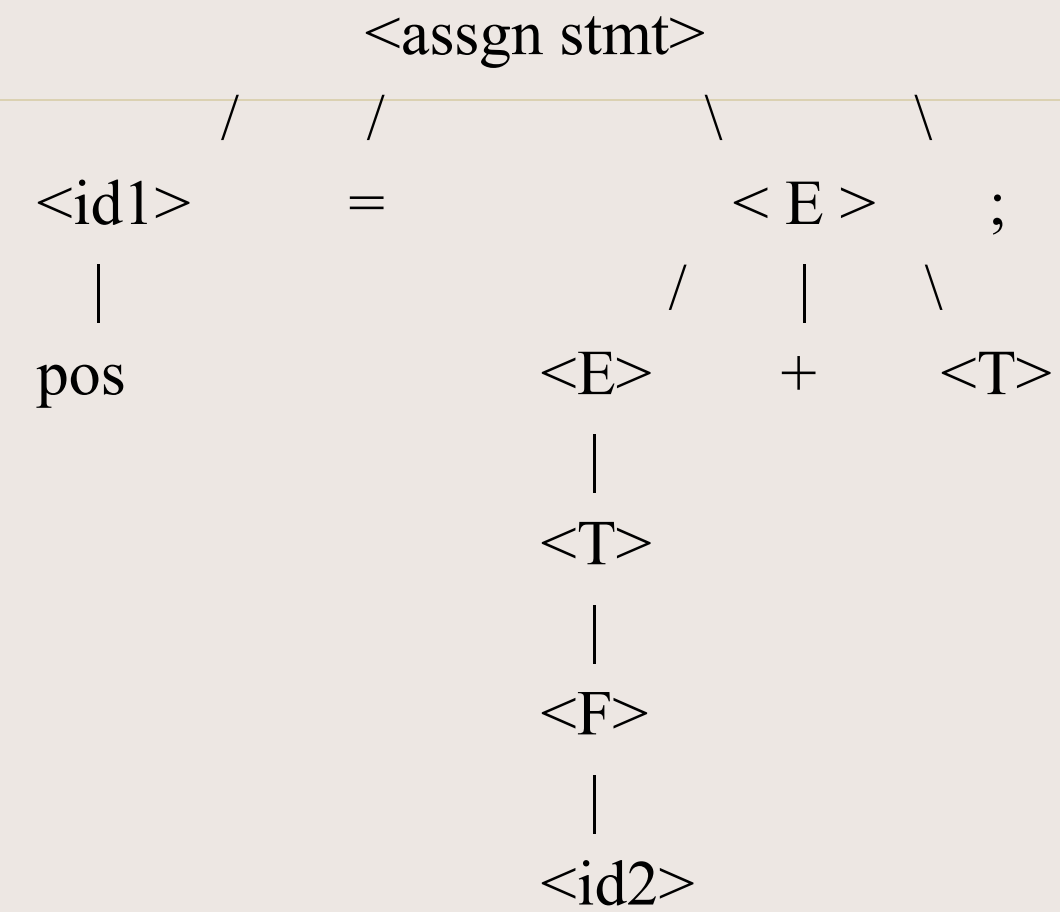
Parse tree



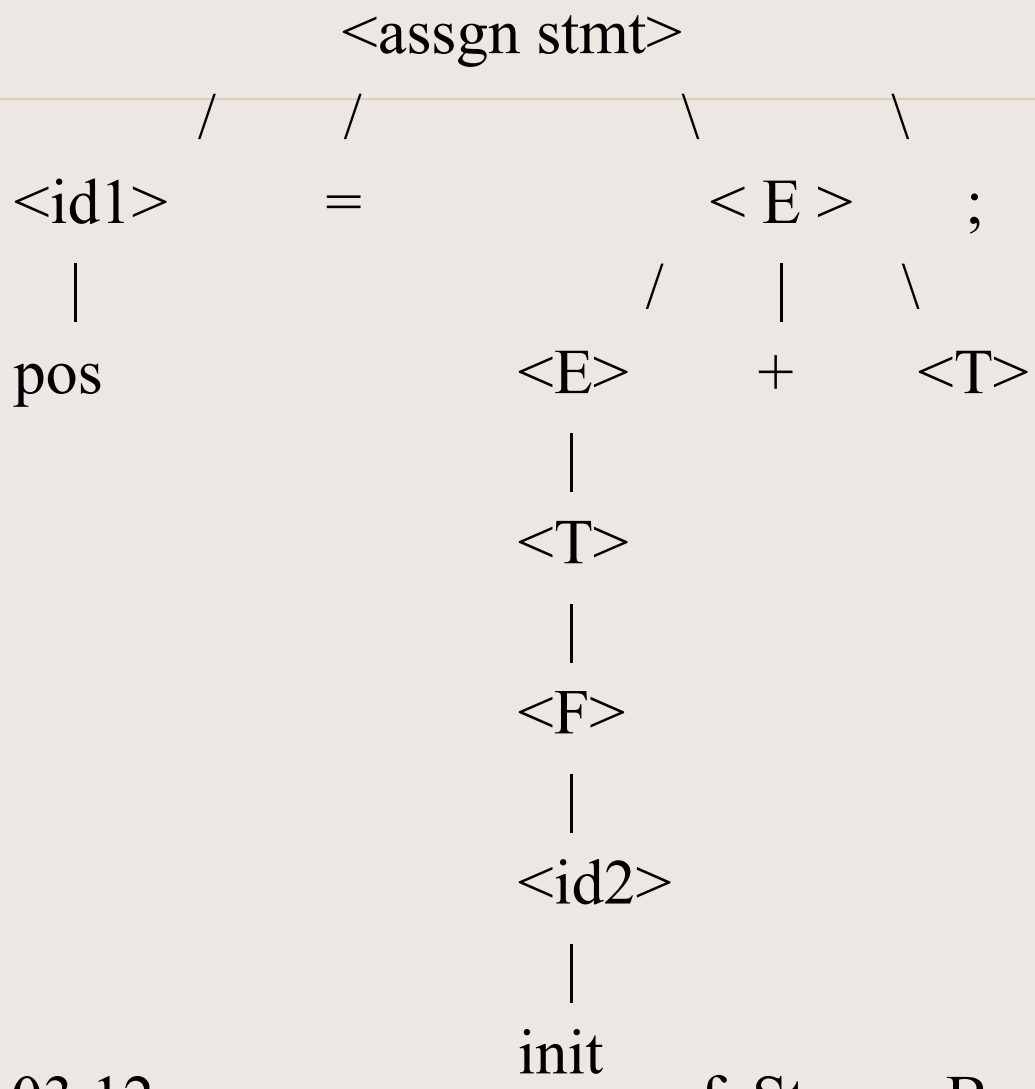
Parse tree



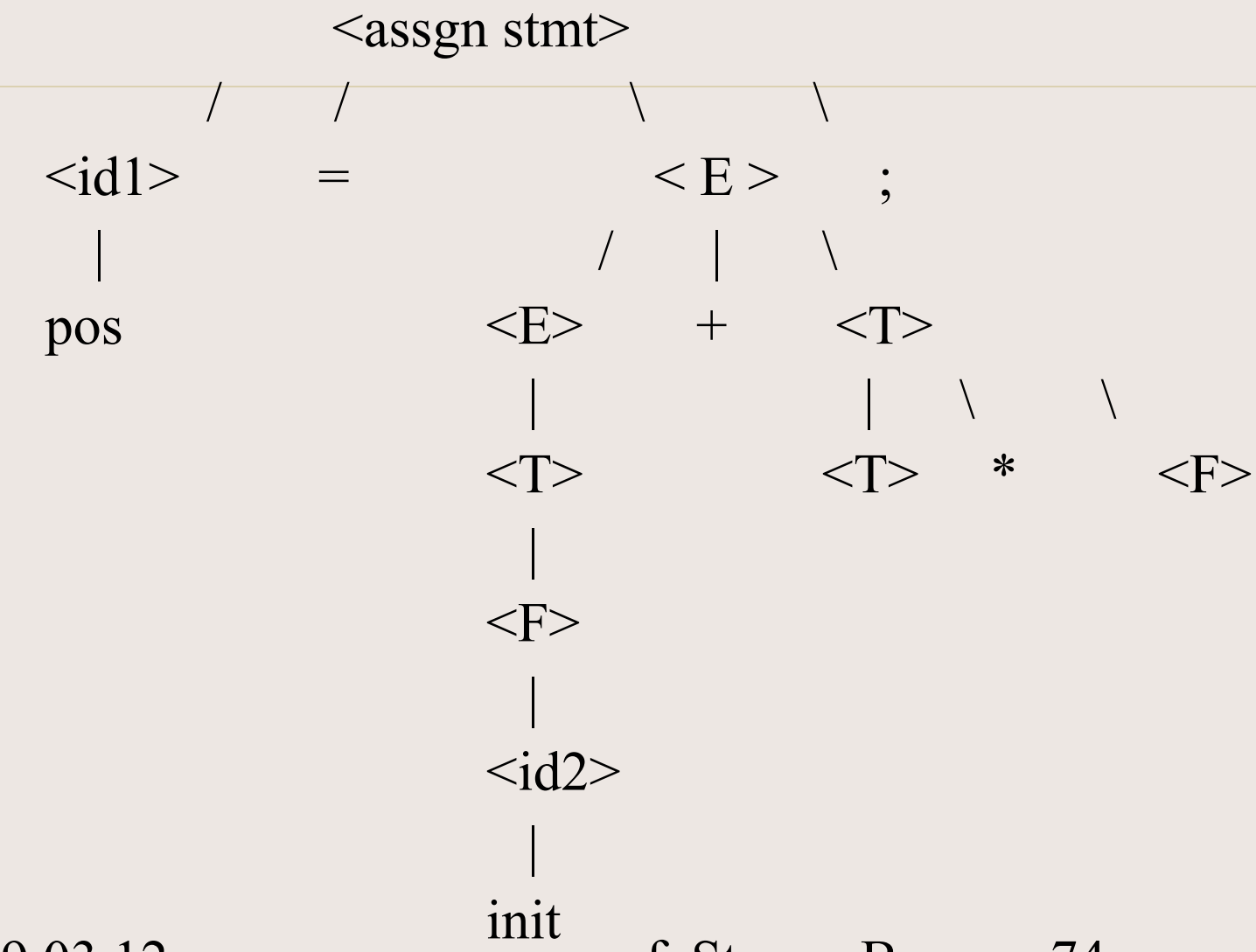
Parse tree



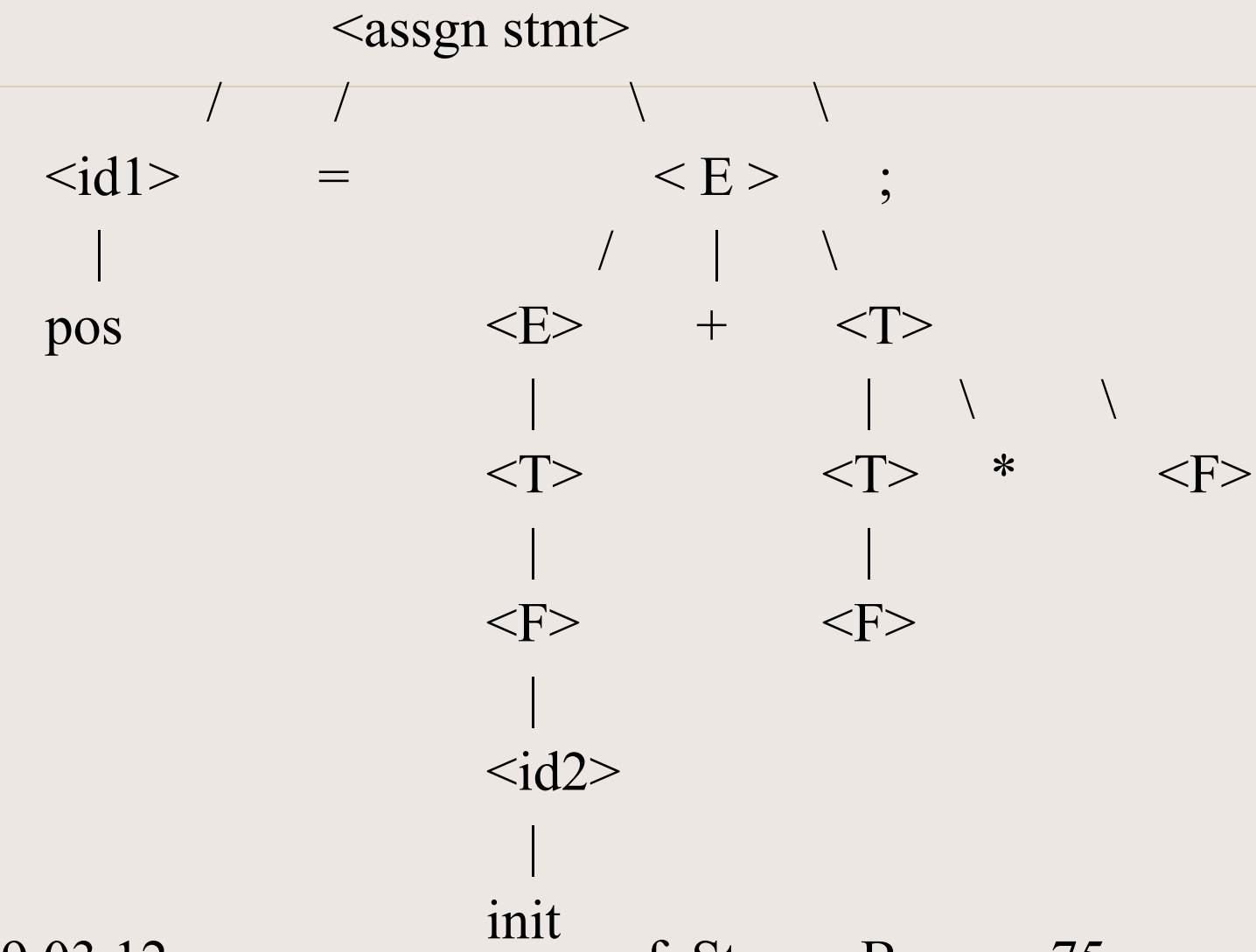
Parse tree



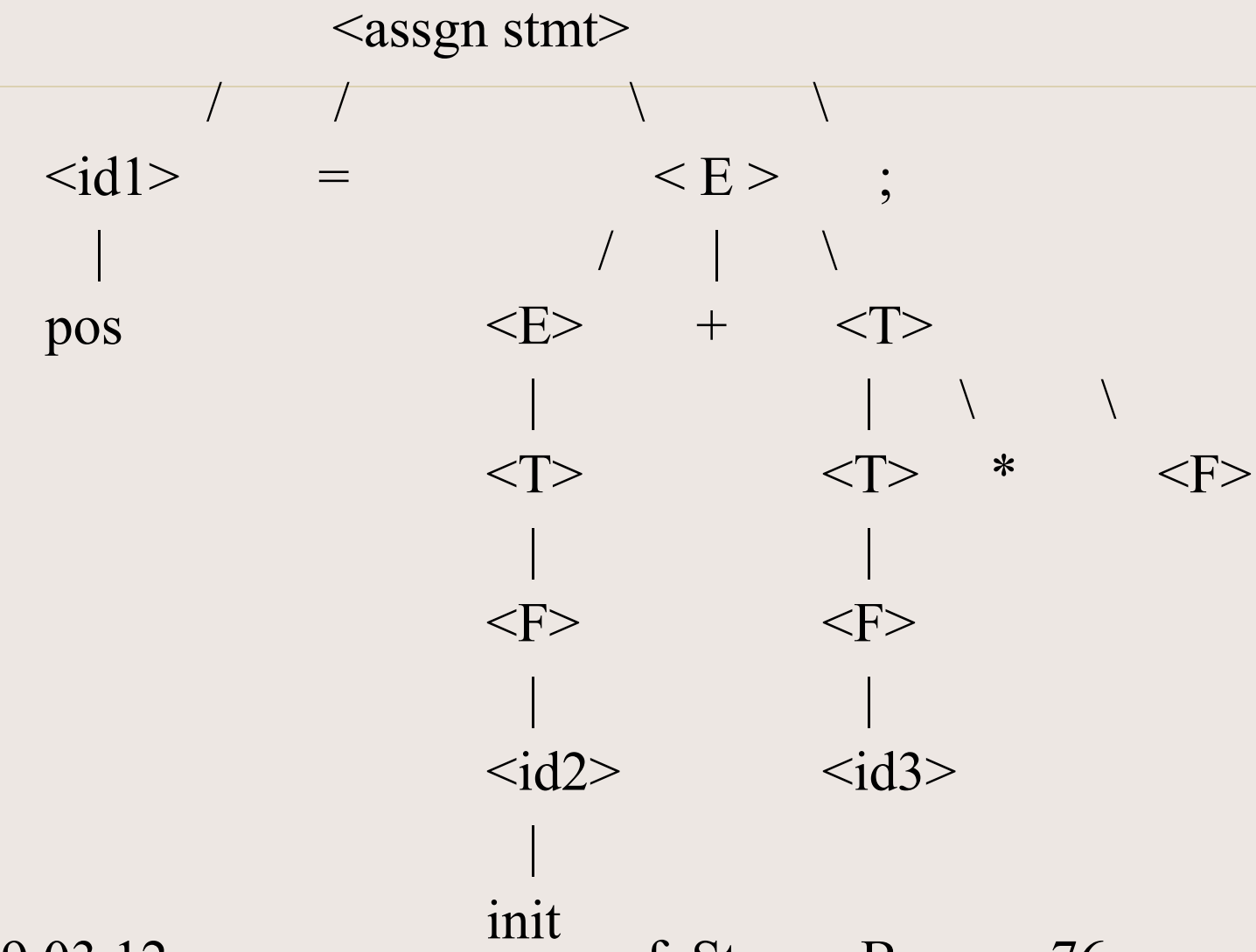
Parse tree



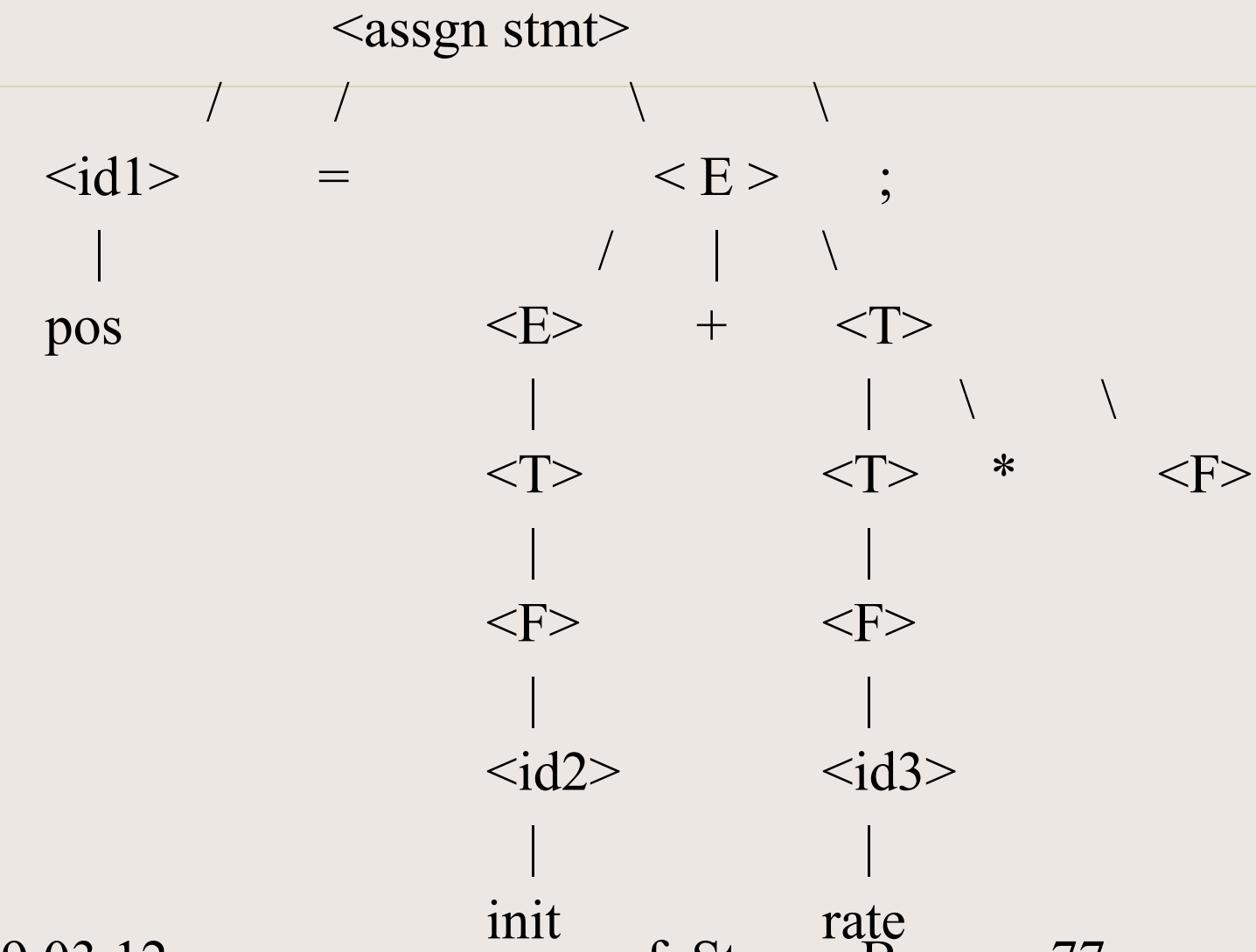
Parse tree



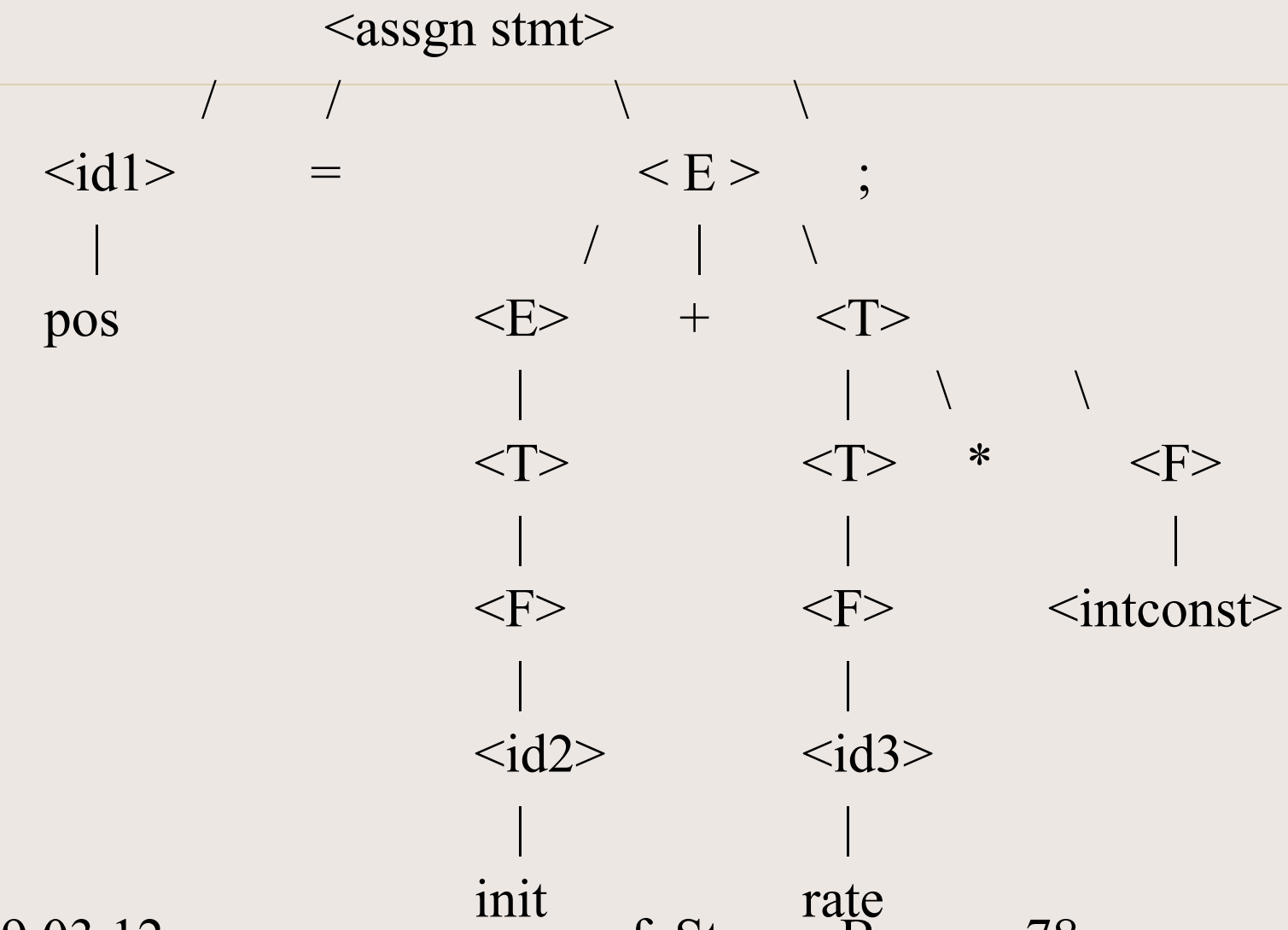
Parse tree



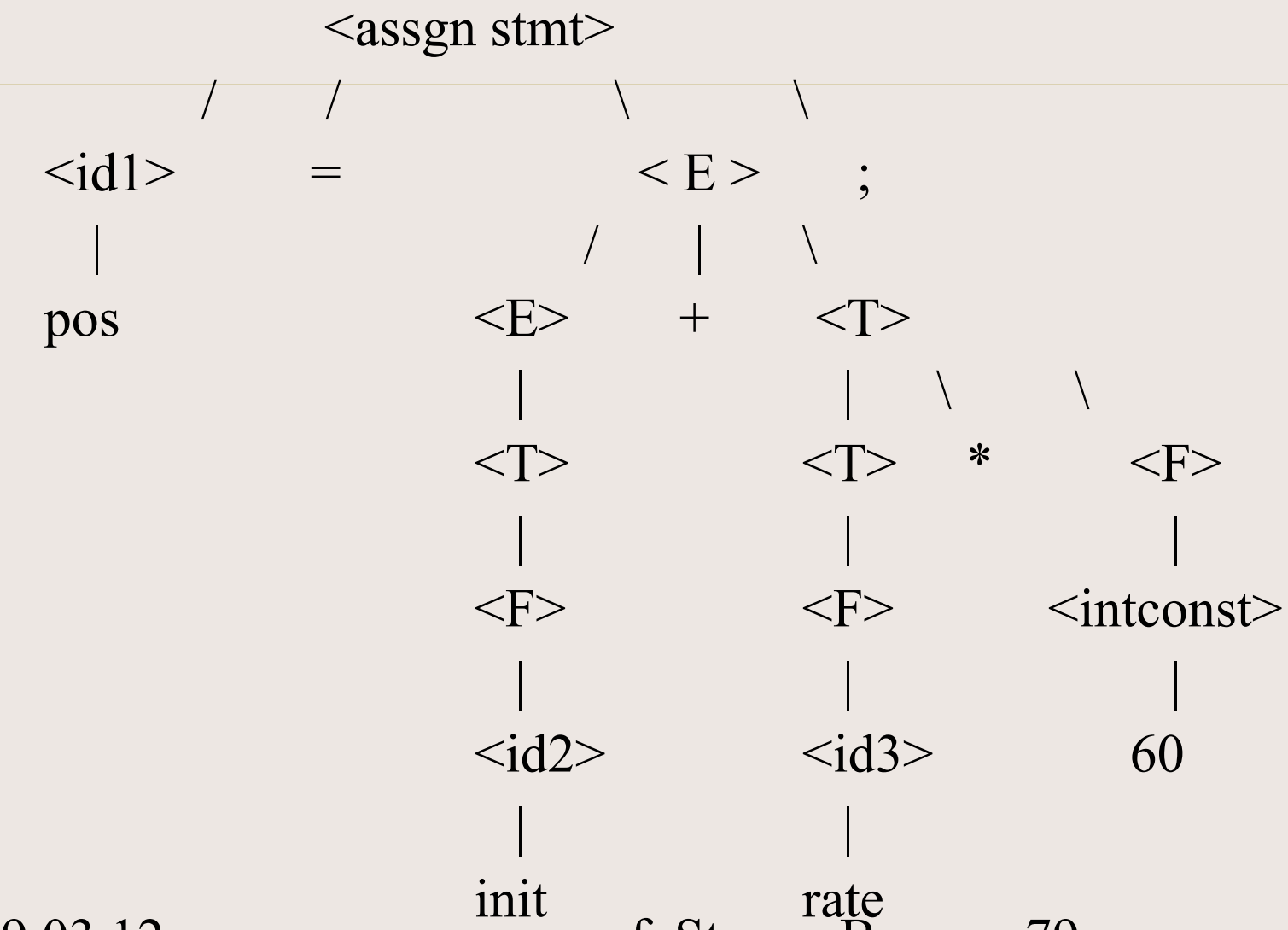
Parse tree



Parse tree



Parse tree



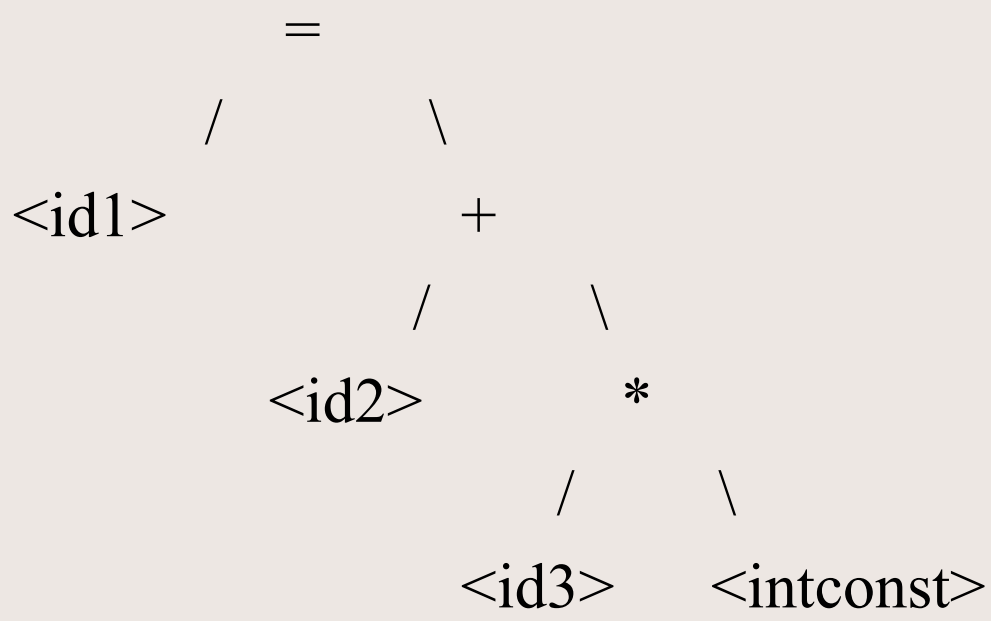
Синтактичен анализ

Syntax tree – компресирано parse tree, в което:

- операциите са възли на дървото
- операндите естествено се съпоставят на възлите операции.

Илюстрация

Syntax tree



Семантичен Анализ

Проверки, които следят дали са:

Дефинирани (декларирани) операнди

Присвоени стойности на операнди

Type checking, т.е операндите допустими ли са по тип.

Вграждане на конвертираща ПП в случай на операнди от смесен тип, както е в примера rate(real), 60 (integer)

Семантичен Анализ

Налице са два операнда от различен тип:

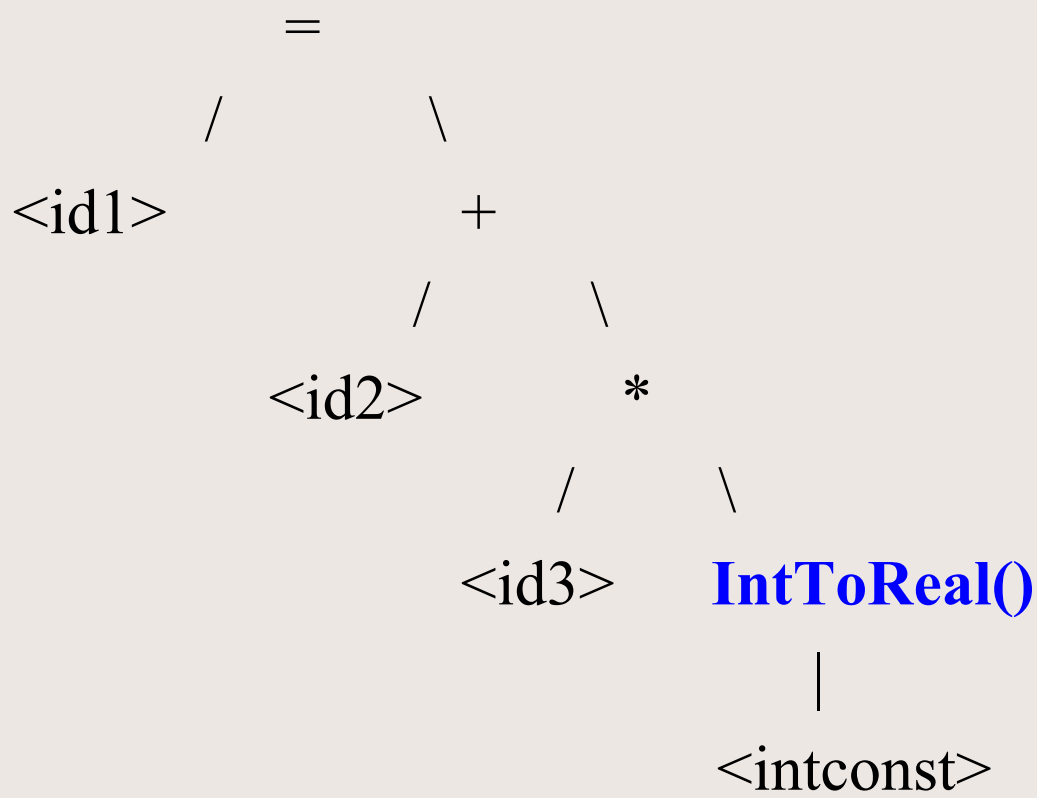
rate (real) 60 (integer)

Уеднаквяване на типа данни за двата операнда се постига с добавяне на допълнителен възел в дървото на разбора - syntax tree

Семантичната ПП ***IntToReal()*** служи като операция за явно преобразуване на цели в реални стойности

Илюстрация – модифицирано дърво на разбор

Семантичен Анализ



Таблици

Статични таблици

- запазени думи
- разделители

Динамични таблици

- потребителски идентификатори
- константи

Таблици

Симв.таблица:	име	атрибути
	`pos`	…
	`init`	…
	`rate`	…

Конст.таблица:	симв.предст.	атрибути
	`60`	…

Обработка на грешки

Три дейности:

- Откриване - Error Detection
- Известяване - Error Reporting
- Възстановяване - Error Recovery

Генерация на междинен код

МК е код за абстрактна машина.

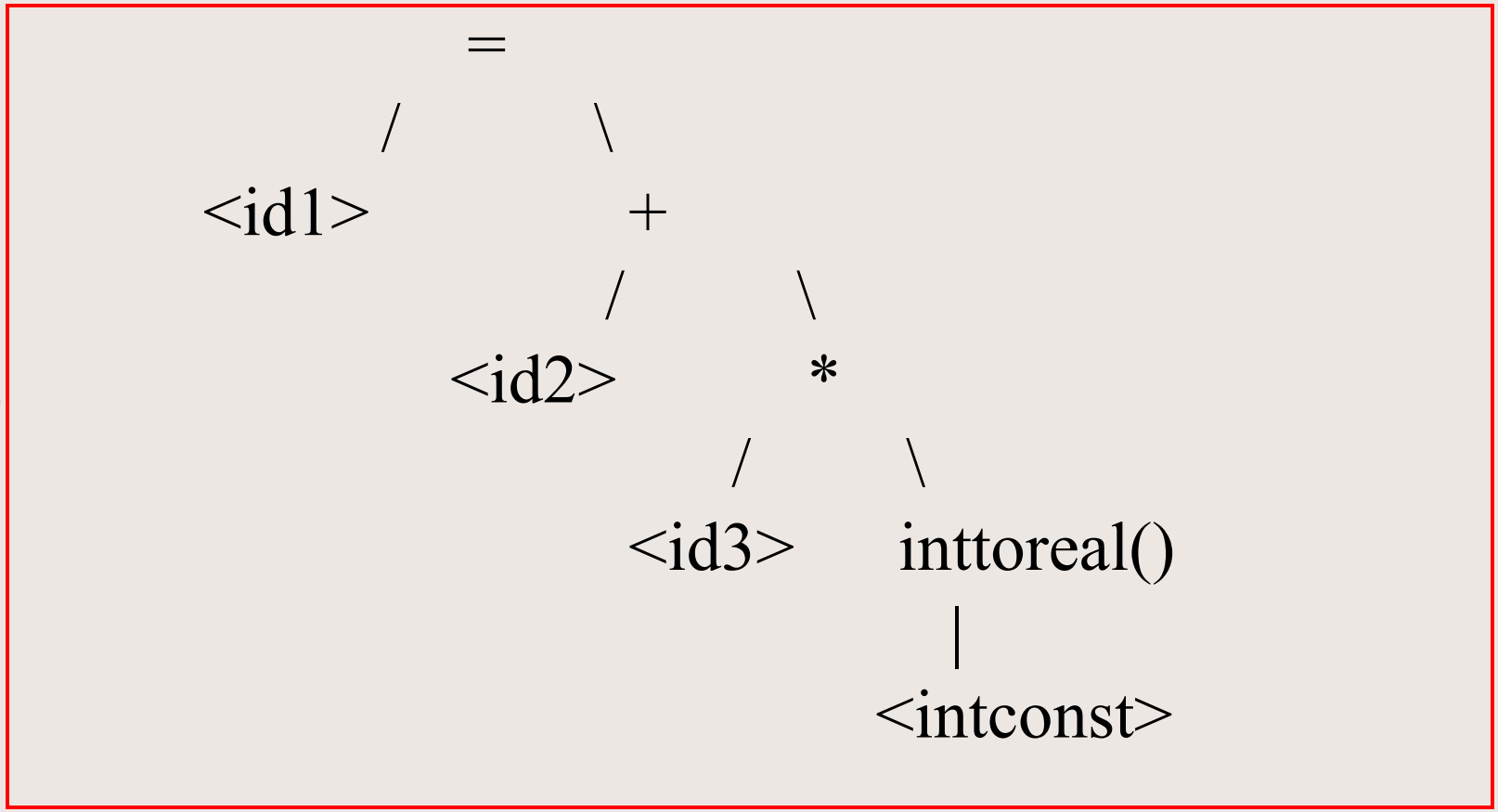
МК да има следните 2 свойства:

- Лесен за генерация
- Лесен за превод в обектен код

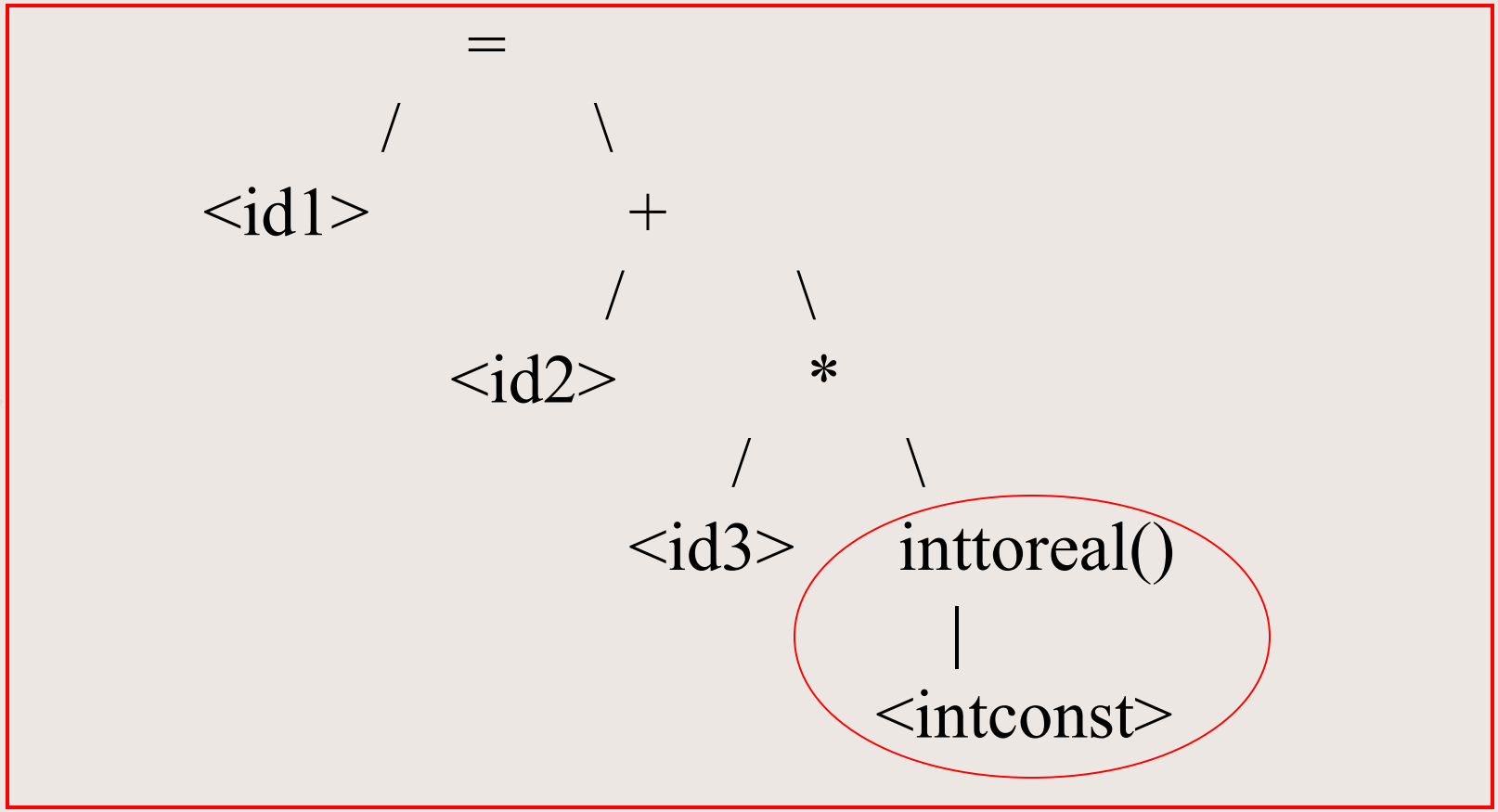
Пример с МК в тетради

(<CodeOp>,<Oprnd1>,<Oprnd2>,<Oprnd3>)

Генерация на междинен код



Генерация на междинен код



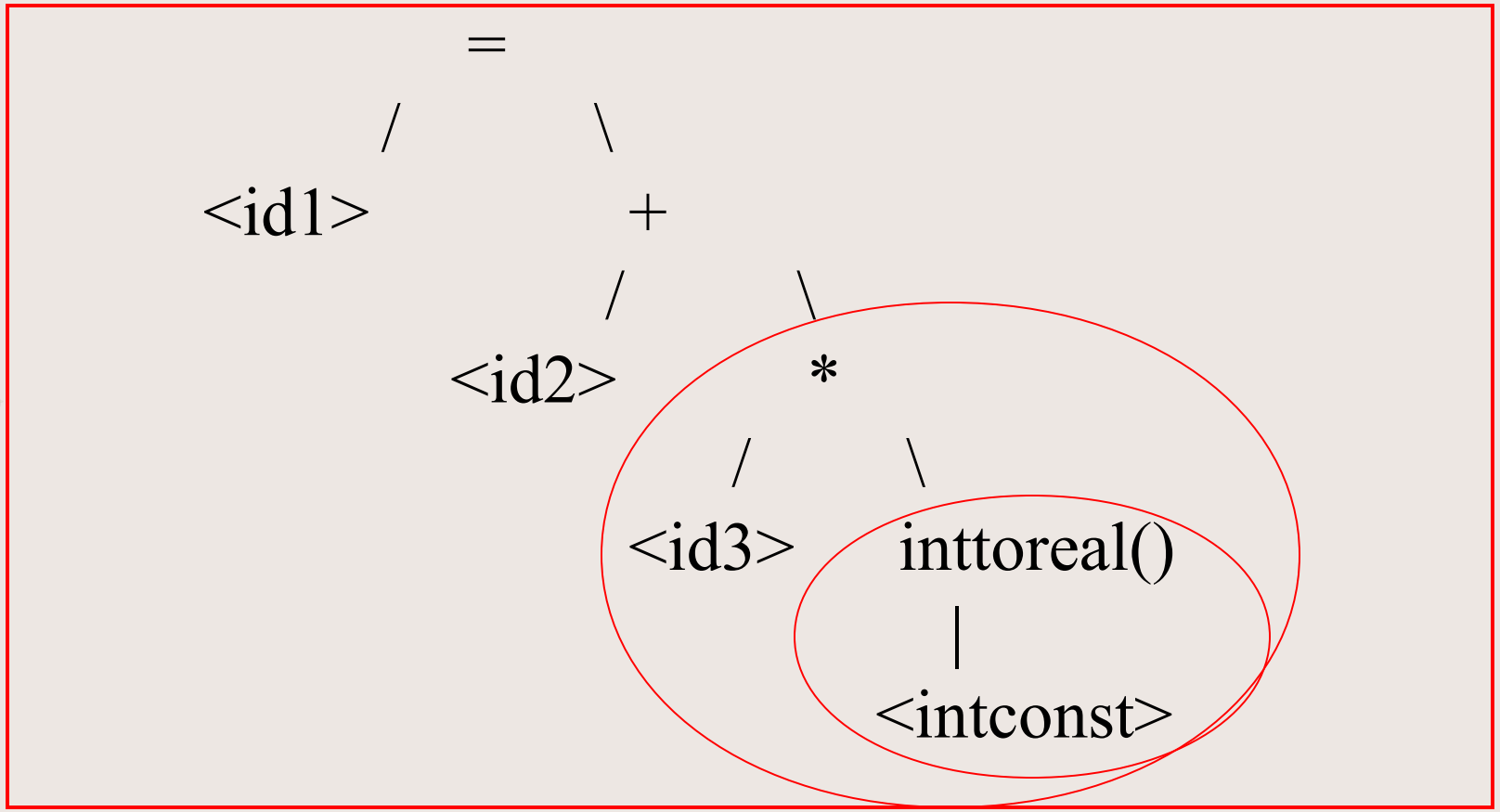
Генерация на междинен код

Всеки възел операция индицира генерация на една тетрада.

<CodeOp>	<Oprnd1>	<Oprnd2>	<Oprnd3>
inttoreal	60	n/a	temp1

```
temp1 = inttoreal(60);
```

Генерация на междинен код



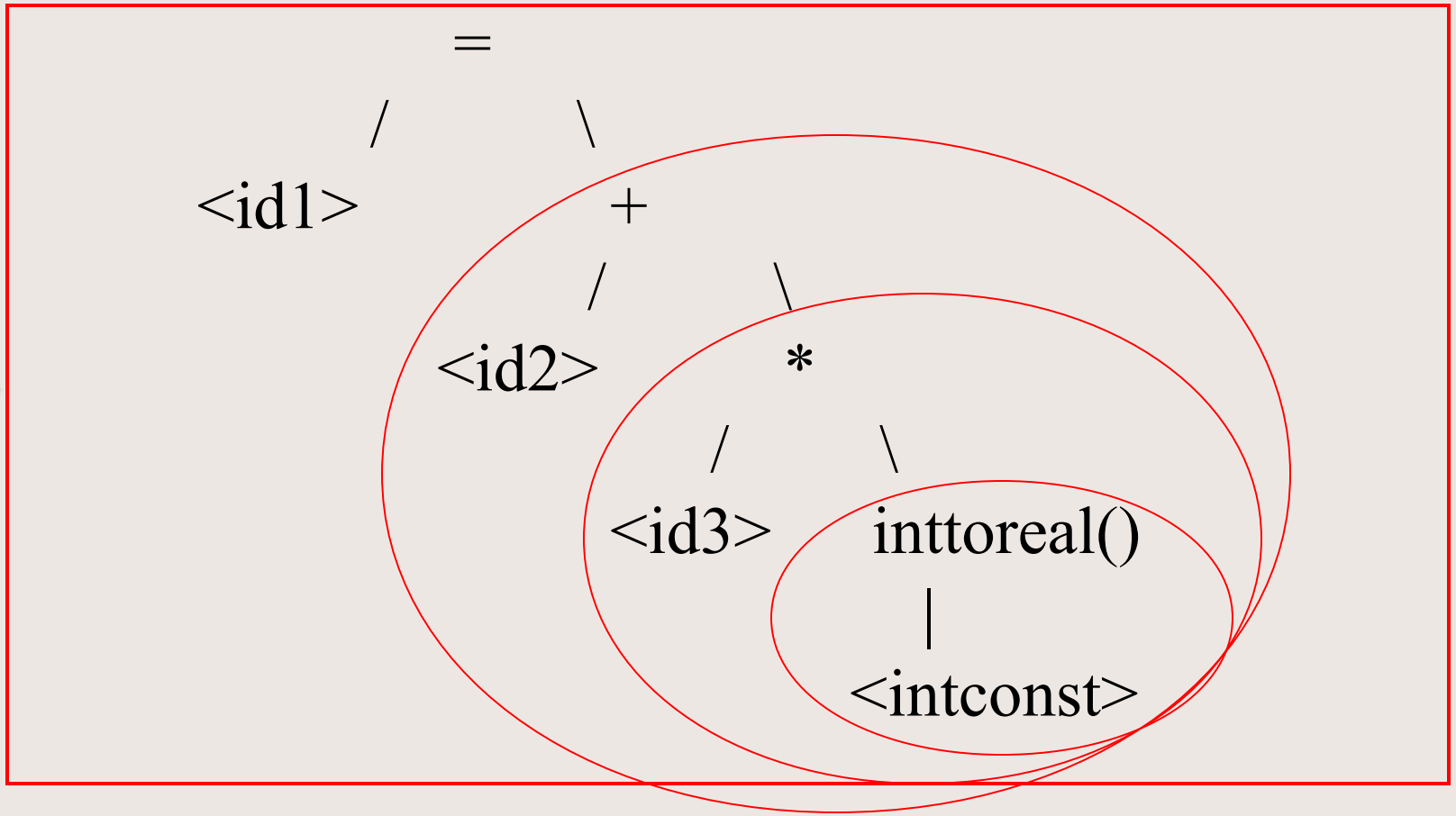
Генерация на междинен код

Всеки възел операция индицира генерация на една тетрада.

<CodeOp>	<Oprnd1>	<Oprnd2>	<Oprnd3>
inttoreal	60	n/a	temp1
*	<id3>	temp1	temp2

```
temp1 = inttoreal(60);
temp2 = <id3> * temp1;
```

Генерация на междинен код



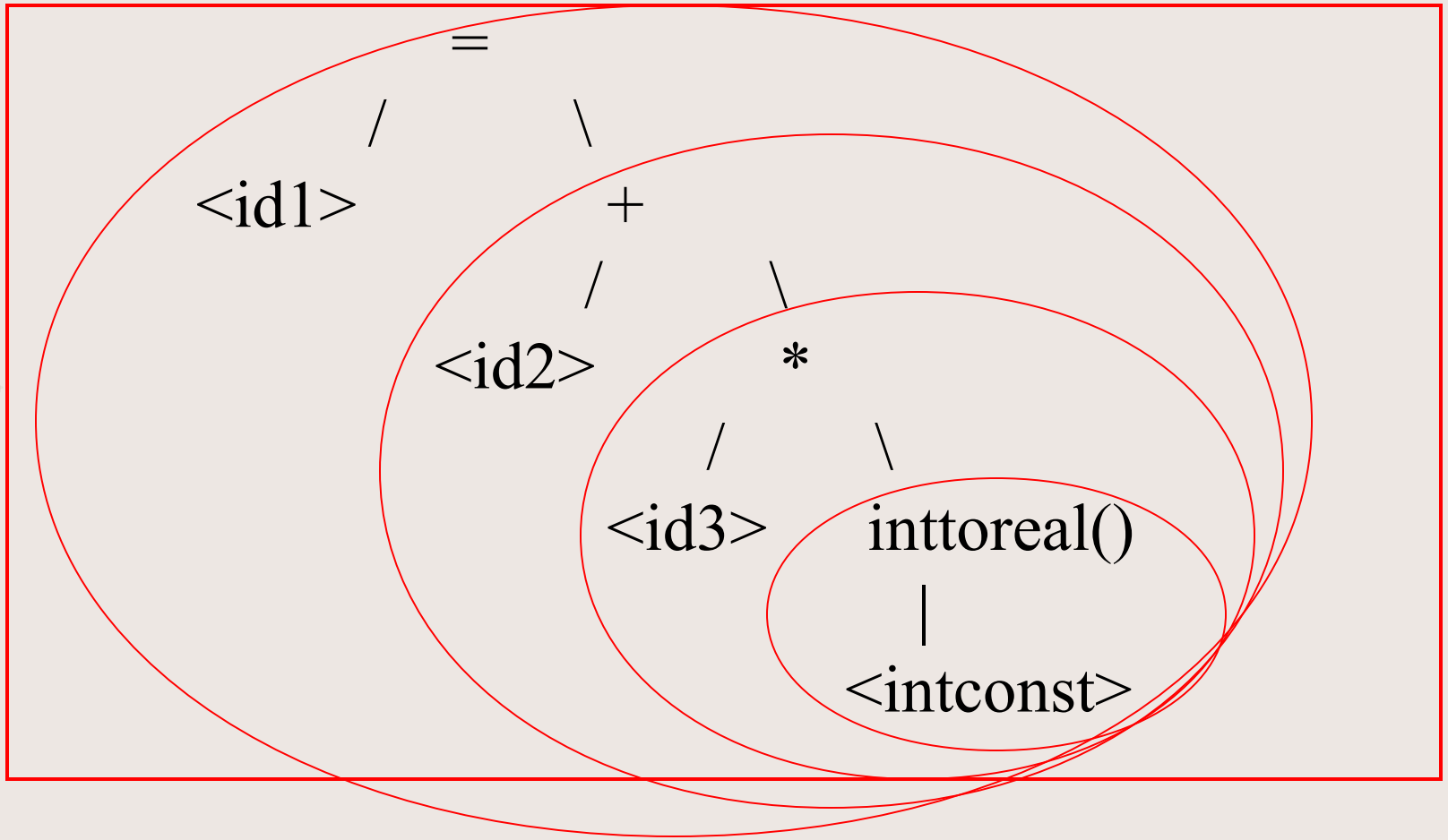
Генерация на междинен код

Всеки възел операция индицира генерация на една тетрада.

<CodeOp>	<Oprnd1>	<Oprnd2>	<Oprnd3>
inttoreal	60	n/a	temp1
*	<id3>	temp1	temp2
+	<id2>	temp2	temp3

```
temp1 = inttoreal(60);
temp2 = <id3> * temp1;
temp3 = <id2> + temp2;
```

Генерация на междинен код



Генерация на междинен код

Всеки възел операция индицира генерация на 1 тетрада.

<CodeOp>	<Oprnd1>	<Oprnd2>	<Oprnd3>
inttoreal	60	na	temp1
*	<id3>	temp1	temp2
+	<id2>	temp2	temp3
=	temp3	na	<id1>

```
temp1 = inttoreal(60);
temp2 = <id3> * temp1;
temp3 = <id2> + temp2;
<id1> = temp3;
```

Оптимизация на код

Идея: вместо да се генерира обектен код за конвертиране от цяло в реално, по-удачно е К. веднъж по време на компилация да конвертира цялата константа 60 в реална константа 60.0.

Това прави излишни семантичната процедура *inttoreal()* и временната променлива *Temp1*.

Резултатът е оптимизиран междинен код. Виж сл. Слайд.

Оптимизация на код

```
temp1 = inttoreal(60)
temp2 = <id3> * temp1
temp3 = <id2> + temp2
<id1> = temp3

temp1 = <id3> * 60.0
<id1> = <id2> + temp1
```

Генерация на код

```
temp1 = <id3> * 60.0
            movf  reg1, <id3>
            mulf  reg1, #60.0
-------------------------------------------------------------
<id1> = <id2> + temp1
            movf  reg2, <id2>
            addf  reg1, reg2
            movf  <id1>, reg1
```

Асемблер
като
Езиков Процесор

ЕП Асемблер

Assembly language is a class of low-level languages used to write computer programs.

- **Assembly language** is a human-readable notation for the machine language used to control a specific computer architecture.
- An **assembler** is a computer program for translating **assembly language** — essentially, a mnemonic representation of machine language — into object code.
- **Machine language** is a pattern of bits encoding machine operations, specific to a given processor.

Assembly language was once widely used for all aspects of programming. Today it is used in limited situations, primarily when direct hardware manipulation or unusual performance issues are involved.

ЕП Асемблер

An **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities.

A program written in assembly language consists of

- a series of *instructions* that correspond to a stream of executable instructions that can be loaded into memory and executed.
- A series of *directives* that control the regime of the assembler operation.

ЕП Асемблер

For example, an x86/IA-32 processor can execute the following binary instruction as expressed in machine language:

- Binary: 10110000 01100001 (Hexadecimal: 0xb061)

The equivalent assembly language representation is easier to remember (more *mnemonic*):

- mov al, 061h

This instruction means:

- Move the hexadecimal value 61 (97 decimal) into the processor register named "al".

The mnemonic "mov" is an *operation code* or *opcode*, and was chosen by the instruction set designer to abbreviate "move." A comma-separated list of arguments or parameters follows the opcode;

ЕП Асемблер

Machine language is built up from discrete *statements* or *instructions*. Depending on the processing architecture, a given instruction may specify:

- Particular registers for arithmetic, addressing, or control functions
- Particular memory locations or offsets
- Particular addressing modes used to interpret the operands

More complex operations are built up by combining these simple instructions, which (in a von Neumann machine) are executed sequentially, or as otherwise directed by control flow instructions.

ЕП Асемблер

Some operations available in most instruction sets include:

- moving
  - set a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
  - move data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
  - read and write data from hardware devices

ЕП Асемблер

Some operations available in most instruction sets include:

- computing
  - add, subtract, multiply, or divide the values of two registers, placing the result in a register
  - perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register
  - compare two values in registers (for example, to see if one is less, or if they are equal)

ЕП Асемблер

Some operations available in most instruction sets include:

- affecting program flow
  - jump to another location in the program and execute instructions there
  - jump to another location if a certain condition holds
  - jump to another location, but save the location of the next instruction as a point to return to (a *call*)

ЕП Асемблер

Теория и практика

АЕ съдържа:

АЕ директиви – команди към Асемблиращата програма, не се превеждат в МЕ команди

АЕ изпълними инструкции – превеждат се в МЕ команди

ЕП Асемблер

Other elements common to most assembly languages include the following:

- **Data definitions.** Additional directives let the programmer reserve storage areas for reference by machine language statements. Storage can typically be initialized with literal numbers, strings, and other primitive data types.
- **Labels.** Data definitions are referenced using names (*labels* or *symbols*) assigned by the programmer, and typically reference constants, variables, or structure elements. Labels can also be assigned to code locations, i.e. subroutine entry points or GOTO destinations. Most assemblers provide flexible symbol management, letting programmers manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler.
- **Comments.** Like most computer languages, comments can be added to assembly source code that are ignored by the assembler.
- **Macros.** Most assemblers have an embedded macro language that generate code or data based on a set of arguments. Macros can be coded by the programmer to avoid repetition, e.g. generating a common data structure. Macros are also supplied by a vendor or manufacturer to encapsulate a particular operation.

ЕП Асемблер

АЕ съдържа:

АЕ директиви – команди към Асемблиращата програма, не се превеждат в МЕ команди

Примери:

Name <име - identifier>

Begin <адрес>

Org <адрес>

End

ЕП Асемблер

АЕ съдържа:

АЕ изпълними инструкции – превеждат се в МЕ команди

Примери:	Мнемоничен формат	МЕ формат
	Ladr <id>	03 xxxx
	Load <id>	06 xxxx
	Ldim <const>	00 xx
	Goto <address>	0a xxxx
	Ifnot <address>	09 xxxx
	If <address>	0c xxxx
	Add	51
	Mul	54
	Store	57
	Nop	5a
	Exit	84

ЕП Асемблер

Две Основни цели:

Превод на мнемо инструкциите в МЕ команди

Превод на символните имена в адреси от паметта (адресно съответствие на имената – операнди и/или етикети)

Изброените задачи се решават в два паса – проблемът **forward reference**

ЕП Асемблер

- Задачи – първи пас
  - Превод на мнемоничните инструкции в МЕ команди
  - Създаване на символна таблица с адресно съответствие
- Задачи – втори пас
  - Запълване на адресните стойности за всички операнди

Асемблер демо: pos=init+rate*60;

```
begin 1000;
        pos = init + rate * 60;
        stop;
begin 2000;
        decl pos, init, rate;
```

Асемблер демо:

`pos=init+rate*60;`

ОПЗ: `pos' init rate 60 * + =`

```
Ladr  pos
Load  init
Load  rate
Ldim  60
Mul
Add
Stor
```

Асемблер демо: pos=init+rate*60;

```
        Name  DemoProg
        Begin   1000h
        Ladr  pos
        Load  init
        Load  rate
        Ldim  60
        Mul
        Add
        Stor
        Exit
        Begin   2000h
Pos     db  1
Init    db  1
Rate    db  1
        End
```

Ac. първи пас: pos=init+rate*60;

```
        Name  DemoProg          Loc    Content
        Begin   1000h           1000
        Ladr  pos               1000   03   xxxx
        Load  init              1003   06   xxxx
        Load  rate              1006   06   xxxx
        Ldim  60                1009   00   3c
        Mul                     100b   54
        Add                     100c   51
        Stor                    100d   57
        Stop                    100e   84
        Begin   2000h           2000
Pos     db  1                   2000   xx
Init    db  1                   2001   xx
Rate    db  1                   2002   xx
        End
```

Асемблер демо: симв. Таблица след първи пас

Символно име	Адрес	Атрибути
DemoProg	1000	Program name
Pos	2000	Size 1 byte
Init	2001	Size 1 byte
Rate	2002	Size 1 byte

Ас. втори пас: pos=init+rate*60;

```
                Name  DemoProg          Loc    Content
                Begin   1000h           1000
                Ladr  pos               1000   03    2000
                Load  init              1003   06    2001
                Load  rate              1006   06    2002
                Ldim  60                1009   00    3c
                Mul                     100b   54
                Add                     100c   51
                Stor                    100d   57
                Exit                    100e   84
                Begin   2000h           2000
Pos             db  1                   2000   xx
Init            db  1                   2001   xx
Rate            db  1                   2002   xx
                End                     2003
```

Благодаря
За
Вниманието