Lecture -1-
Introduction to Compiler Structure

أ.م.د. سهاد مال الله

أ.م.د. عبير طارق مولود

أ.م. علاء نوري

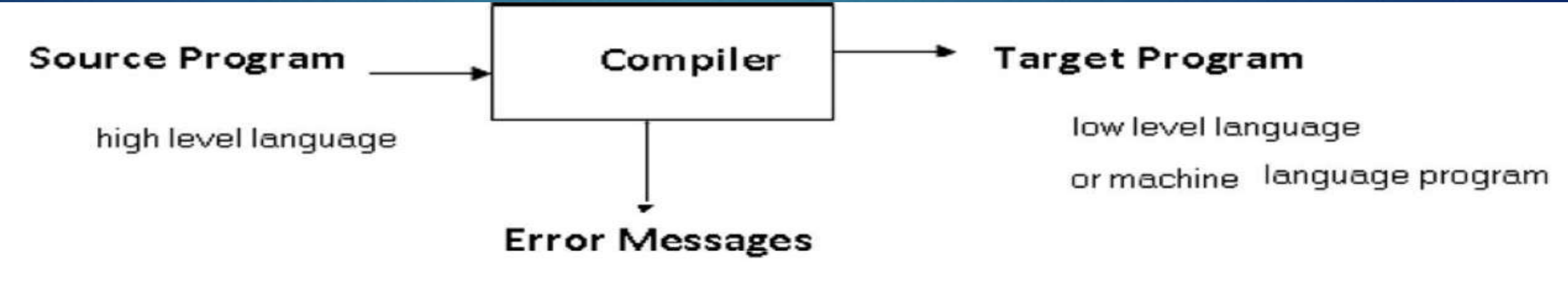
تصميم المترجمات/الكورس الثاني/كل الافرع

قسم علوم الحاسوب/الجامعة التكنولوجية

2021-2020

Compiler

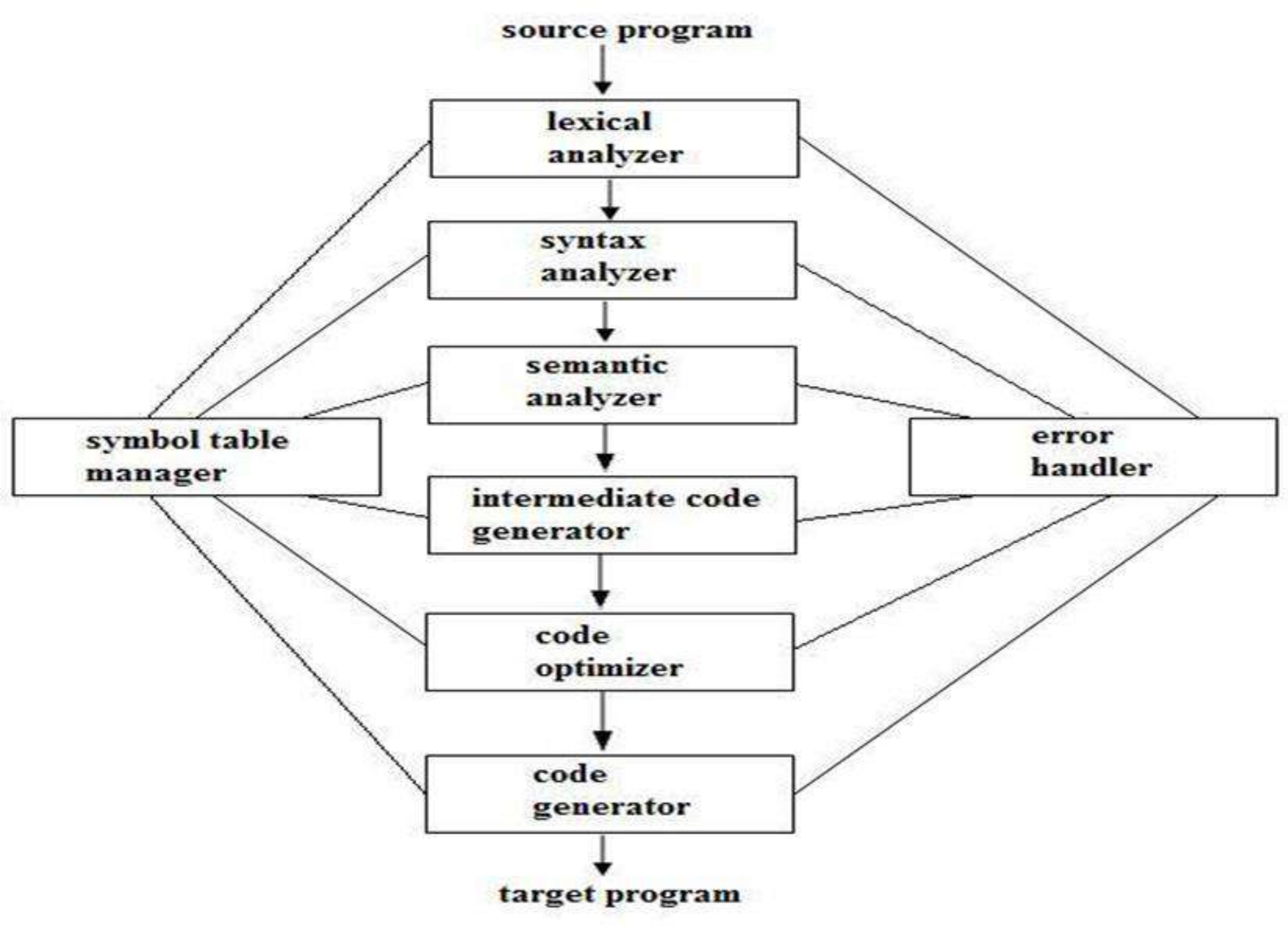
Is a program (translator) that reads a program written in one language, (the source language) and translates into an equivalent program in another language (the target language). A translator, which transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler.**



The time at which the conversion of the source program to an object program occurs is called (compile time) the object program is executed at (run time).

Compiler structure:

A compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in figure bellow:



1- lexical analysis

The lexical analyzer is the first stage of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

2- Syntax analysis (parsing)

The syntax analysis (or parsing) is the process of determining if a string of tokens can be generated by grammar. Every programming language has rules that prescribe the syntactic structure of well-formed programs. Syntax Analyzer takes an out of lexical analyzer and produces a large tree

3- Semantic analysis

The semantic analysis phase checks the source program for semantics errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements. Semantic analyzer takes the output of syntax analyzer and produces another tree.

4- Intermediate code generation

Generate an explicit intermediate representation of the source program. This representation should have two important properties,

1- it should be easy to produce

2- easy to translate into the target program.

5- Code Optimization

Attempts to improve the intermediate code so that faster running machine code will result.

6- code generation

Generates a target code consisting normally of machine code or an assemble code. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are each translated in to a sequence of machine instructions that perform the same task.

Symbol table management :

Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as type (integer, real, etc.). The data structure used to record this information is called symbolic table.

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various source language constructs. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code.

Symbol table operations:

1-insert(s,t) : this function is to add a new name to the table

2-Lookup(s) : returns index of the entry for string s

3-Delete a name or group of names from the tables.

4-Update

5-Search

Error handler:

Is called when an error in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can produced.

Types of errors

The syntax and semantic phases usually handle a large fraction of errors detected by compiler.

1. Lexical error: The lexical phase can detect errors where the characters remaining in the input do not form any token of the language . few errors are discernible at the lexical level alone ,because a lexical analyzer has a very localized view of the source program. Example : If the string fi is encountered in a C program for the first time in context:

fi (a== f(x)….

A lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function name. since fi is a valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

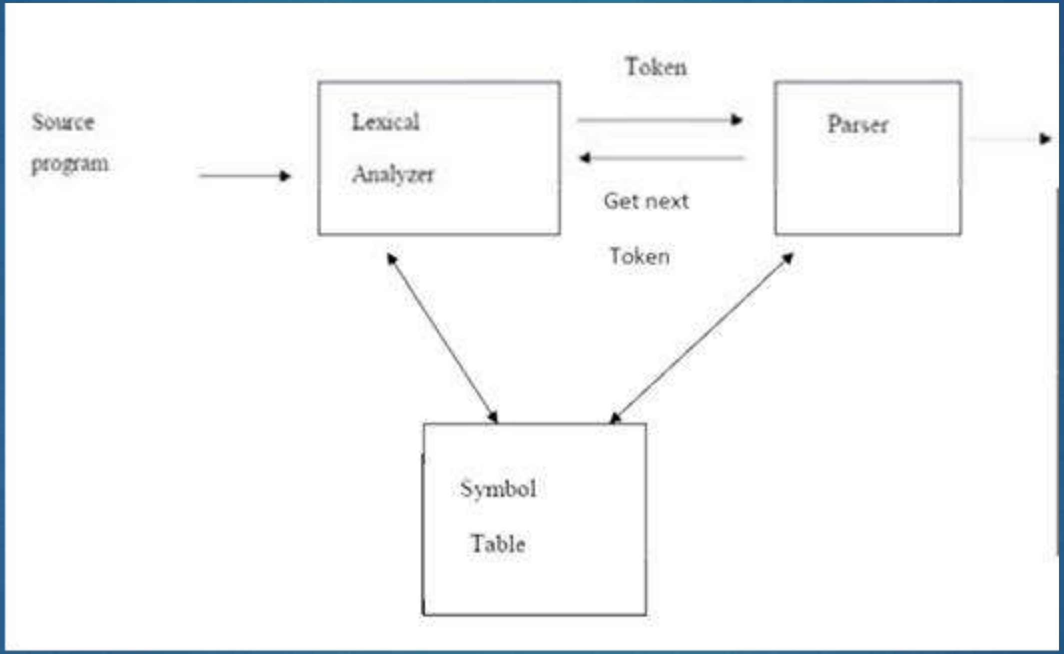
2- syntax error: The syntax phase can detect Errors where the token stream violates the structure rules (syntax) of the language.

3- semantic error: During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.

4- runtime error. It occurs during the implementation of the program, such as mathematical errors, such as dividing by zero or exceeding the permissible limits of the matrix and other errors

Lexical Analyzer

The lexical analyzer is the first phase of compiler. The main task of lexical Analyzer is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc.



Interaction of lexical analyzer with parser

Token is a string of character ended with space ex: for i := 1 to 4 do (7 tokens)
For token1 ,   i token 2 , := token 3 , 1 token 4 , to token 5 , 4 token 6 , do token 7

secondary tasks of lexical analyzer

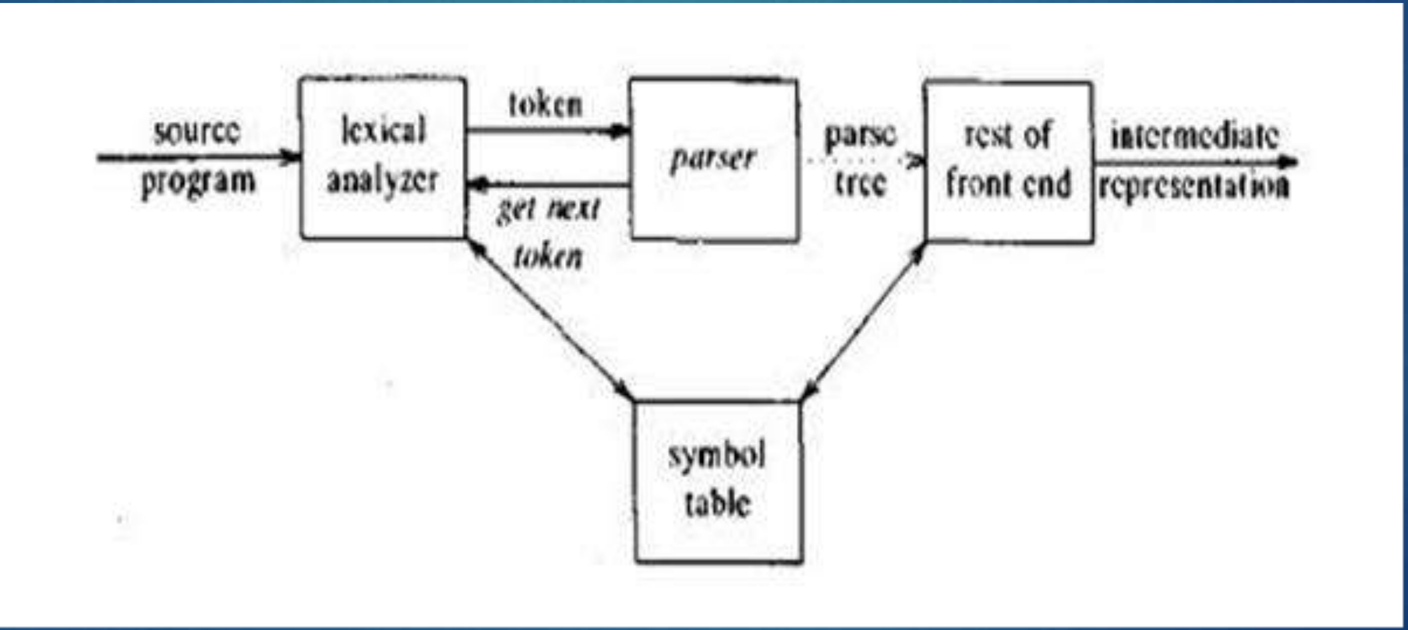
1- stripping out from the source program comments and white space in the form of blank, tab, and new line characters.

2- correlating error messages from the compiler with the source program.

lexical analyzers are divided into a cascade of two phases, the first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Syntax Analysis

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source program. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



Position of parser in Compiler model

The methods commonly used in compilers are classified as being either Top-down or bottom up. As indicated by their names, Top down parsers build parse trees from the top (root) to the bottom (leaves) and work up to the root. In both cases, the input to the parser is scanned from left to right, one symbol at time.

Thank you for Listening

Lecture -2-
Top Down parsing method & problems

أ.م.د. سهاد مال الله

أ.م.د. عبير طارق مولود

أ.م. علاء نوري

تصميم المترجمات/الكورس الثاني/كل الافرع

قسم علوم الحاسوب/الجامعة التكنولوجية

2020-2021

Top down parser

In this section there are basic ideas behind top-down parsing and show how constructs an efficient non- backtracking form of top-down parser called a predictive parser.

Top down parsing can be viewed as attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

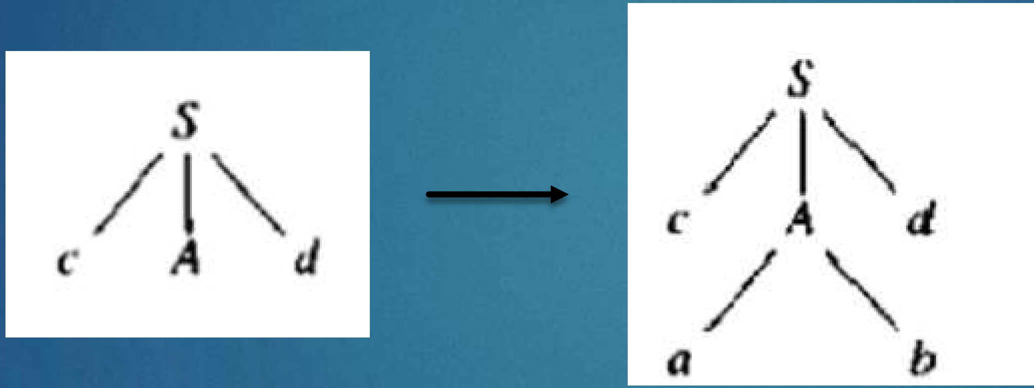
The following grammar requires backtracking:

Backtracking example:

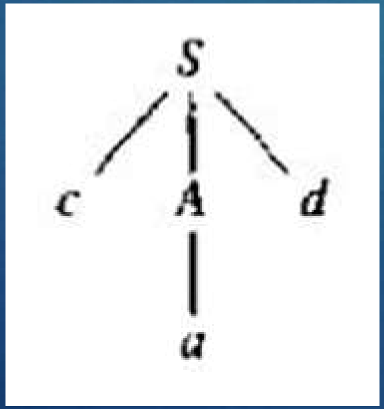
$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

the input string $w = cad$.

Step 1:-



Step 2:-



problems of grammar

1- Ambiguity:

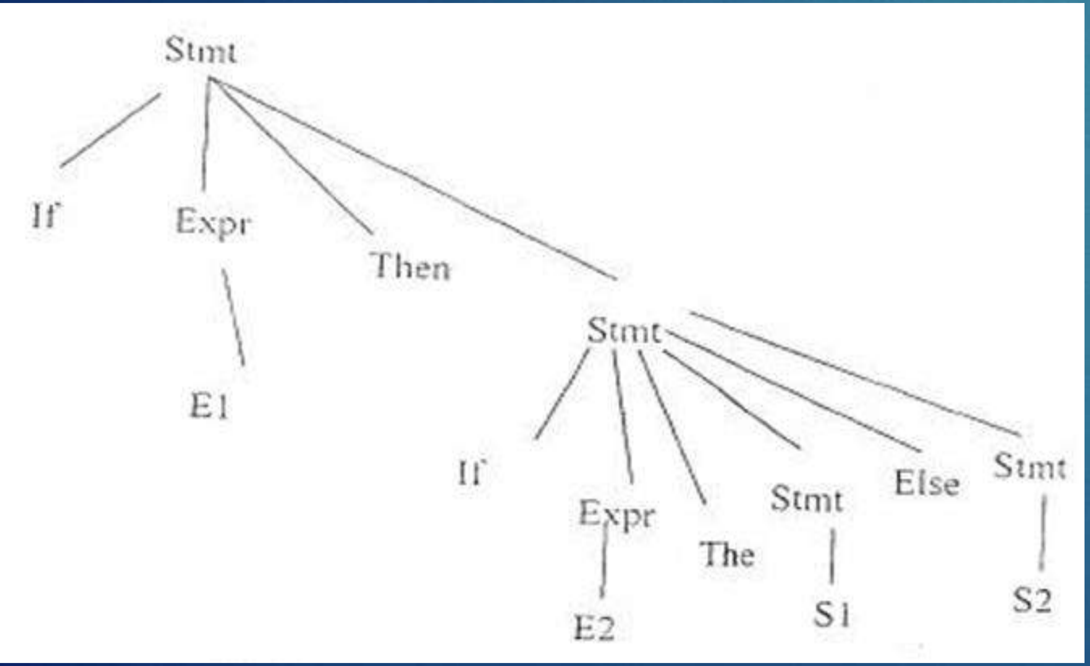
A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar is one that produces more than one leftmost or more than one right most derivation for the same sentence. For certain types of parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Example:-

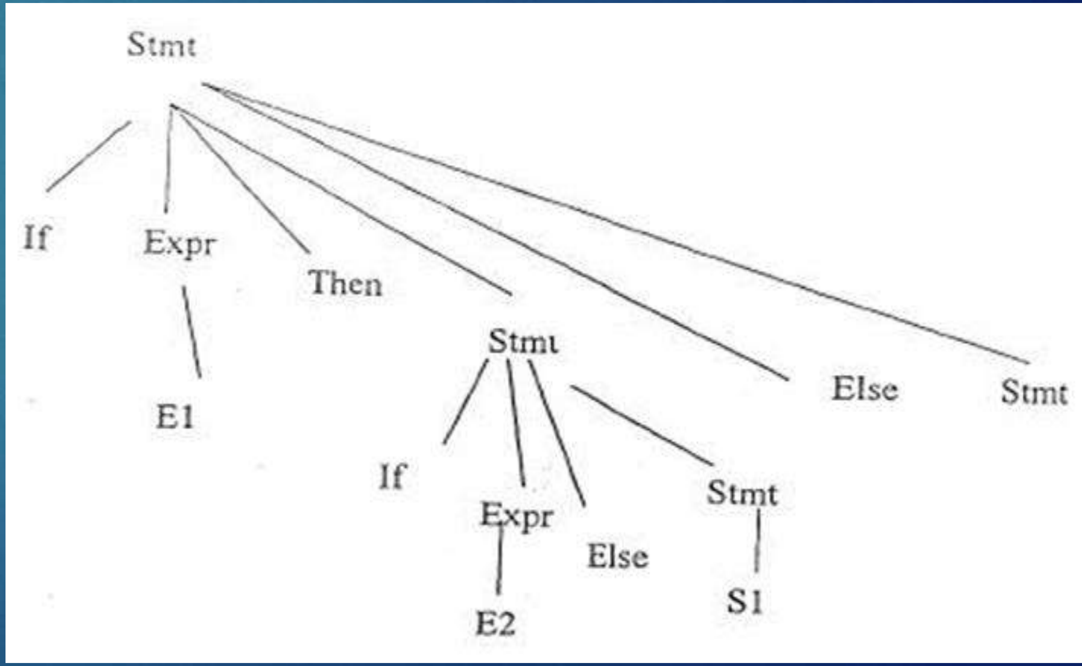
string If E1 then if E2 then S1 else S2

Has the two parse trees shown below:

TREE 1



TREE 2



2- Left Recursion

A grammar is left recursion if it has a nonterminal A, such that there is a derivation **A** $\longrightarrow$ **Aα | β** For some string α. Top-down parsing methods cannot handle left recursion grammars, so a transformation that eliminates left recursion is needed.

Example:-

$$\mathbf{A} \longrightarrow \mathbf{A\alpha} \mid \boldsymbol{\beta}$$

جزء المشكلة   الجزء الخالي من المشكلة

Solution:

تكون بداية الحل بالجزء الخالي من المشكلة

$$\mathbf{A} \longrightarrow \boldsymbol{\beta}\mathbf{A'}$$

$$\mathbf{A'} \longrightarrow \boldsymbol{\alpha}\mathbf{A'} \mid \lambda$$

Example 1 : **Consider the following grammar for arithmetic expressions.**

$$E \longrightarrow E+T \mid T$$
$$T \longrightarrow T*F \mid F$$
$$F \longrightarrow (E) \mid id$$

Solution:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \lambda$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \lambda$$
$$F \rightarrow (E) \mid id$$

General rule:

$$A \longrightarrow A\alpha1 \mid A\alpha2 \mid \ldots \mid A\alpha n \mid \beta1 \mid \beta2 \mid \ldots\ \beta n$$

where no βi begins with an A. then, we replace the A -productions by

$$A \longrightarrow \beta1A' \mid \beta2A' \mid \ldots \mid \beta nA'$$
$$A \longrightarrow \alpha1A' \mid \alpha2A' \mid \ldots \mid \alpha nA' \mid \lambda$$

Example 2 :

$$S \longrightarrow SAb \mid SBa \mid B$$
$$A \longrightarrow bb$$
$$B \longrightarrow Ac \mid cb$$

Solution:

$$S \longrightarrow BS'$$
$$S' \longrightarrow AbS' \mid BaS' \mid \lambda$$
$$A \longrightarrow bb$$
$$B \longrightarrow Ac \mid cb$$

Example 3:

$$\mathbf{A \longrightarrow AA \mid Ab}$$

Solution:

$$\mathbf{A' \longrightarrow AA' \mid bA' \mid \lambda}$$

ملاحظة:- في المثال الثالث يتم اهمال قاعدة الجزء الخالي من المشكلة وذلك لكونه غير موجود والانتقال مباشرة لحل جزء المشكلة

Indirect Left Recursion :-

Consider the following example:

$$S \longrightarrow Aa \mid b$$

$$A \longrightarrow Ac \mid Sd \mid \lambda$$

The non-terminal S is left recursion because:

$$S \longrightarrow Aa \longrightarrow Sda$$

But is not immediately left recursion.

3- Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice. For example, if we have the two productions:

Example1:

$$A \longrightarrow \alpha\,\beta 1 \;|\; \alpha\,\beta 2$$

ملاحظة: يتم تطبيق القانون التالي:
المشترك من الاطراف
المتبقي من الاطراف

Solution:

$$A \longrightarrow \alpha A'$$
$$A' \longrightarrow \beta 1 \,|\, \beta 2$$

Example 2:

$\text{Stmt} \longrightarrow$ if Expr then Stmt else Stmt | if Expr then Stmt | other

Solution:

$\text{Stmt} \longrightarrow$ if Expr then Stmt else Stmt | if Expr then Stmt | other

$\text{Stmt} \longrightarrow$ if Expr then Stmt Stmt’ | other
$\text{Stmt'} \longrightarrow$ else Stmt | λ

Example 3:

$S \longrightarrow$ af | ac | aa | bb | bd | b | xx

Solution:

$S \longrightarrow$ aA’ | bB’ | xx
$A' \longrightarrow$ f | c | a
$B' \longrightarrow$ b | d | λ

Double Left Factoring :-

Consider the following grammar:

S $\longrightarrow$ aabca | aabd | aab | ccb

This grammar is double left factoring:

Step 1:

S $\longrightarrow$ aabca | aabdx | aab | aabd | cc

الاولوية للعدد الاكثر من الطرق المشتركة

Solution:

S $\longrightarrow$ aabS’ | cc

S’ $\longrightarrow$ ca | dx | λ | d

Step 2:

S $\longrightarrow$ aabS’ | cc

S’ $\longrightarrow$ dS” | ca | λ

S” $\longrightarrow$ x | λ

Quiz:

Consider the following grammar *G*.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \text{id} \mid \text{id}\,(\,) \mid \text{id}\,(L)$$
$$L \rightarrow E \,;\, L \mid E$$

1. Eliminate left recursion:
2. Eliminate left factoring.

Thank you

Lecture -3-
First and Follow

أ.م.د. سهاد مال الله
أ.م.د. عبير طارق مولود
أ.م. علاء نوري

تصميم المترجمات/الكورس الثاني/كل الافرع

قسم علوم الحاسوب/الجامعة التكنولوجية

2020-2021

Introduction

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible.

First Definition

Define the FIRST(α) to be the set of terminals that begin the strings derived from α, and the FOLLOW(A) for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form.

To compute FIRST(x) for all grammar symbols x, apply the following rules until no more terminals or ε can be added to any first set:

1- If x is terminal, then FIRST(x) is {x}.

2- If X→ a ; is a production, then add a to FIRST(X) and
If X → ∈ ; is a production, then add ∈ to FIRST(X).

3- If X is nonterminal and X→Y1,Y2…Yi ; is a production, then add FIRST(Y1) to FIRST(X).

4-a- for (i = 1; if Yi can derive epsilon ∈; i++)
b- add First(Yi+1) to First(X)
If Y1 does not derive ∈ , then we add nothing more to FIRST(X), but if Y1→ ∈ , then we add FIRST(Y2) and so on .

examples

Example1:

FIRST (terminal) = {terminal }

$S \rightarrow aSb \mid ba \mid \in$

FIRST (a) ={a}

FIRST (b) ={b}

2-FIRST(non terminal) = FIRST (first char)

FIRST (S)= {a,b, ∈}

Example 2:

$S \rightarrow aSb \mid X$

$X \rightarrow cXb \mid b$

$X \rightarrow bXZ$

$Z \rightarrow n$

	First
S	a , c , b
X	c , b
Z	n

Example 3:

$S \rightarrow bXY$
$X \rightarrow b \mid c$
$Y \rightarrow b \mid \epsilon$

	First
S	b
X	b , c
Y	b , ∈

Example 4:

$S \rightarrow ABb \mid bc$
$A \rightarrow \epsilon \mid abAB$
$B \rightarrow bc \mid cBS$

	First
S	b , a , c
A	∈ , a
B	b , c

Example 4:-

$X \rightarrow ABC \mid nX$
$A \rightarrow bA \mid bb \mid \epsilon$
$B \rightarrow bA \mid CA$
$C \rightarrow ccC \mid CA \mid cc$

First

X	n , b , c
A	b , ϵ
B	b , c
C	c

Follow Difinition

FOLLOW(A) for all non terminals A, is the set of terminals that can appear immediately to the right of A in some sentential form S → aAxB... To compute Follow, apply these rules to all nonterminals in the grammar:

1- Place $ in FOLLOW(S) , where S is the start symbol and $ is the input right end marker.
FOLLOW(START) = {$}

2- If there is a production X→ α Aβ , then everything in FIRST(β) except for ε is placed in FOLLOW(A).
i.e. FOLLOW(A) = FIRST(β)

3- If there is a production X→ α A, or a production X→ α Aβ , where FIRST(β) Contains ε (β → ∈), then everything in FOLLOW(X) is in FOLLOW(A).
i.e. : FOLLOW(A)= FOLLOW(X)

$$X \rightarrow \alpha\ A\ \beta$$

حالات ال β:-

1- البيتا غير فارغة اذن نقوم بأخذ ال first لها ونضعها في قيم قيم ال follow لل nonterminal المعني

2- البيتا فارغة اذن نقوم بأخذ قيم ال follow لل nonterminal الموجود ما قبل السهم ونضعها ضمن قيم ال follow لل nonterminal المعني

Example 1:

S → aSb | X
X → cXb | b
X → bXZ
Z → n

	First	Follow
S	a , c , b	$, b
X	c , b	b , n , $
Z	n	b , n , $

Example 2:

$S \rightarrow bXY$

$X \rightarrow b \mid c$

$Y \rightarrow b \mid \in$

	First	Follow
S	b	\$
X	b , c	b , \$
Y	b , ∈	\$

ملاحظة مهمه:- يمنع منعا باتا وجود (∈) ضمن قيم ال follow

Example 3:

$S \rightarrow ABb \mid bc$
$A \rightarrow \epsilon \mid abAB$
$B \rightarrow bc \mid cBS$

	First	Follow	
S	b , a , c	$, b , c , a	+ follow(B)
A	∈ , a	b , c	
B	b , c	b , c , a	

ملاحظة: في حالة احتياجنا الى قيم follow لم يتم حسابها نقوم بكتابة ملاحظة وبعد استخراجها نكتبها بدل الملاحظة ونقوم بمسح الملاحظة

Example 4:

$X \rightarrow ABC \mid nX$

$A \rightarrow bA \mid bb \mid \in$

$B \rightarrow bA \mid CA$

$C \rightarrow ccC \mid CA \mid cc$

	First	Follow
X	n , b , c	$
A	b , ∈	b ,c,$
B	b , c	c
C	c	b , $, c

+ follow(B) + follow (C)

Quiz:

Consider the following grammar:

$S \rightarrow bSX \mid Y$
$X \rightarrow XC \mid bb$
$Y \rightarrow b \mid bY$
$C \rightarrow ccC \mid CX \mid cc$

- **Find First and Follow**

Note that there is a left recursion and left factoring you must trying to solve these problems before finding first and follow.

Thank you

Lecture -4-
Predictive Parsing Method

أ.م.د. سهاد مال الله
أ.م.د. عبير طارق مولود
أ.م. علاء نوري

تصميم المترجمات/الكورس الثاني/كل الافرع
قسم علوم الحاسوب/الجامعة التكنولوجية
2020-2021

Predictive Parsing Method

Is a Top-down parsing method.

There are four steps to parse using this method:

1. Eliminate left recursion and left factoring if exist.
2. Find first and follow.
3. Construct the parsing table
4. Construct the stack.

Example: consider the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Solution

Step1: Eliminate left recursion and left factoring if it founded:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Step2: find first and follow

	First	Follow
E	(, id	$,)
E’	+, ϵ	$,)
T	(, id	+ ,) , $
T’	*, ϵ	+ ,) , $
F	(, id	+ , * ,) , $

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

ملاحظة:في حالة ال left recursion فأن قيمة ال follow لأي X’, X تكون متشابهه

Step 3: construct the parsing table:

ملاحظة مهمة: يتم الاعتماد في بناء جدول الاعراب على قيم ال first وفي حالة وجود empty في قيم ال first يتم الانتقال الى قيم ال follow ويتم ذكر انه هذا ال nonterminal كان يؤدي الىempty في قيم ال follow هذه.

Non Terminal	Input Symbol					
	id	+	*	(	)	$
E	E→ TE'			E→ TE'		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

	First	Follow
E	(, id	$,)
E’	+, ϵ	$,)
T	(, id	+ ,) , $
T’	*, ϵ	+ ,) , $
F	(, id	+ , * ,), $

Step 4: construct the Stack:
The moves made by predictive parser on input

id + id * id

ملاحظة: يتم اعطاء الكلمة ال (word) sentence من قبل الاستاذ وبعد اعرابها داخل ال stack تكون اما accept او reject

Stack	Input	Output
$ E	**id + id*id $**	
$\$E'T$	**id + id*id $**	$E\rightarrow TE^{-}$
$\$E'T'F$	**id + id * id $**	$T^{-} \rightarrow FT^{-}$
$\$E'T'\ \cancel{id}$	$\cancel{id}$ **+ id * id $**	$F \rightarrow id$
$\$\ E'T'$	**+ id * id $**	
$\$\ E'$	**+ id * id $**	$T^{-} \rightarrow \epsilon$
$\$E'T\ \cancel{+}$	$\cancel{+}$ **id * id $**	$E^{-} \rightarrow +TE^{-}$
$\$\ E'T$	**id * id $**	
$\$\ E'T'F$	**id * id $**	$T \rightarrow FT^{-}$
$\$E'T'\ \cancel{id}$	$\cancel{id}$ *** id $**	$F \rightarrow id$
$\$\ E'T'$	*** id $**	
$\$\ E'T'F\cancel{*}$	$\cancel{*}$ **id $**	$T^{-} \rightarrow * FT^{-}$
$\$\ E'T'F$	**id $**	
$\$E'T'\ \cancel{id}$	$\cancel{id}$ **$**	$F \rightarrow id$
$\$\ E'T'$	**$**	
$\$\ E'$	**$**	$T^{-} \rightarrow \epsilon$
$	**$**	$E^{-} \rightarrow \epsilon$

Non Terminal	Input Symbol					
	id	**+**	*	**(**	**)**	**$**
E	$E\rightarrow TE'$			$E\rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (\epsilon)$		

ملاحظة:نقوم دائما داخل ال stack بمقاطعة قمة ال stack مع بداية الكلمة ويكتب ما بعد السهم بتسلسل عكسي

The sentence is accept

LL(1) grammars:

A grammar whose parsing table has no multiply-defined entries is said to be LL(1). The first "L" in LL(1) indicates the reading direction (left-to-right), the second "L" indicates the derivation order (left), and the "1" indicates that there is a one-symbol or look ahead at each step to make parsing action decisions.

Example: Consider the following grammar

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

	FIRST	FOLLOW
S	i, a	\$, e
S’	e, ϵ	\$, e
E	b	t

الطريقة السريعة: لمعرفة هل ال grammar هو LL1 او لا نقوم بالبحث عن ال empty داخل قيم ال first لكل ال nonterminals وبعد ايجادها نقوم بمقارنة قيم ال first مع قيم ال follow لهذا الnonterminal فأذا كانت هناك قيم مشتركة اي يوجدterminal واحد او عدد من terminals مشترك مابينهم فيعتبر NOT LL1

The grammar is NOT LL1

الطريقة الثانية بأستخدام جدول الاعراب وكما يلي:

Non Terminal	Input Symbol					
	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtss'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

The grammar is NOT LL1

Thank you

Lecture -6-

SLR Parser

أ.م.د. سهاد مال الله
أ.م.د. عبير طارق مولود
أ.م. علاء نوري

تصميم المترجمات/الكورس الثاني/كل الافرع
قسم علوم الحاسوب/الجامعة التكنولوجية
2020-2021

LR parser

This section presents an efficient bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. These technique is called LR parsing; the L is for .Leftright scanning of the input, the R for constructing a Rightmost derivation in reverse. This method present three techniques for construct an LR parsing table for grammar . The first method, called simple LR (SLR), is easiest to implement. But the least powerful. The second method, called Canonical LR, is the most powerful and will work on a very large class of grammars and the most expensive. The third method, called look ahead LR (LALR), is intermediate in power and cost between the SLR and the Canonical LR methods.

SLR Parser

This method of parsing is the weakest of three in terms of the number of grammar for which it succeeds, but it is easiest to implement this parsing method there are four basic steps:

1. Find first & follow.

2. Find set of item.

3. Find parsing table.

4. Check the sentence (parse the input) using stack.

Example:

consider the grammar:

E → E+T | T

T → T*F | F

F → (E)| id

1. Find first and follow

	First	Follow
E	(, id	$,) , +
T	(, id	$,) , + ,*
F	(, id	$,) , +, *

قبل البدء بالحل نقوم اولا بتهيئة ال grammar للاعراب وذلك عن طريق القيام بالخطوات التالية

1. اضافة production القبول ويكون موقعه دائما دائما كأول production في ال grammar وبالشكل التالي

$$E' \rightarrow .E$$

2. ترقيم ال production داخل ال grammar بالارقام (r1,r2,r3,.....) ويتم كتابة .acc وهي مختصر كلمة accept على production حالة القبول الذي تم اضافته من قبلنا

acc.
E' → .E

r1 r2
E → E+T | T

r3 r4
T → T*F | F

r5 r6
F → (E)| id

بعد ان تمت عملية تهيئة ال grammar نقوم بالخطوة الرئيسية الثانية للاعراب وهي بناء مجموعة ال items

لبناء مجموعة ال items نقوم بأتباع الخطوات المهمة التالية:

1. دائما نبدأ بترقيم ال item من الصفر اي I0,I1,I2,I3,I4,I5,I6
2. نقطة البداية او الانطلاق في بناء ال items تكون من Item 0
3. لبناء مجموعة ال item نقوم بتطبيق القانون المهم التالي وبالتسلسل نفسه:

نقوم بفحص الطرق المتشابهة (في نفس ال item)

نقوم بفحص ما بعد النقطة (Closure) في ال item الجديدة التي تم كتابتها

مع وجود ملاحظة مهمه خاصة بالقانون انه هذا القانون دائما يتم تطبيقه ضمن نفس ال item ولا يجوز ان ننتقل الى غير item وكذلك يتم تطبيقه فقط على ال production التي لم يتم تأشيرها.

4. حالات النقطة (closure) الخاصة بالاعراب:

اذا كان ما بعد النقطة nonterminal نقوم بأستدعاء كل ال productions التي تخص هذا ال nonterminal من ال grammar الاصلي.

اذا كان ما بعد النقطة terminal في هذه الحالة نقطع ونبدأ ب item جديدة

5. لا نقوم بتكرار نفس ال production داخل نفس ال item مع ملاحظة انه التكرار يكون اذا تشابه شكل ال production وموقع النقطة .

$$\left.\begin{array}{l} E \rightarrow E+.T \\ E \rightarrow .E+T \end{array}\right\} \text{غير متشابهة لاختلاف موقع النقطة}$$

6. Item zero يتم بنائها بالاعتماد فقط على فحص ما بعد النقطة وغير مشروط وجود كل ال productions الخاصة بال grammar الاصلي فيها فهذا خاص بحالات النقطة

7. عند وصول النقطة الى نهاية ال production نقوم بتأشيره ب √ دلاله على اكتمال اعرابه ونكتب رقم ال production على السهم

2. Find set of I.

i0:

$E' \rightarrow .E$ √
$E \rightarrow .E+T$ √
$E \rightarrow .T$ √
$T \rightarrow .T*F$ √
$T \rightarrow .F$ √
$F \rightarrow .(E)$ √
$F \rightarrow .id$ √

i1: acc.

$E' \rightarrow E.$ √
$E \rightarrow E.+T$ √

i2:

$E \xrightarrow{r2} T.$ √
$T \rightarrow T.*F$ √

i3:

$T \xrightarrow{r4} F.$ √

i4:

$F \rightarrow (.E)$ √
$E \rightarrow .E+T$ √
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

تكرار

i5:

$F \xrightarrow{r6} id.$ √

i6:

$E \rightarrow E+.T$ √
$T \rightarrow .T*F$ √
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

تكرار

i7:

$T \rightarrow T*.F$ √
$F \rightarrow .(E)$
$F \rightarrow .id$

تكرار

i8:

$F \rightarrow (E.)$ √
$E \rightarrow E.+T$

i9:

$E \xrightarrow{r1} E+T.$ √
$T \rightarrow T.*F$

i10:

$T \xrightarrow{r3} T*F.$ √

i11:

$F \xrightarrow{r5} (E).$ √

$E' \rightarrow E$	acc
$E \rightarrow E+T$	r1
$E \rightarrow T$	r2
$T \rightarrow T*F$	r3
$T \rightarrow F$	r4
$F \rightarrow (E)$	r5
$F \rightarrow id$	r6

ملاحظة مهمة: نقوم بفحص الطرق المتشابهة في التكرار لمرة واحدة فقط ويهمل باقي التكرار

Thank you

Lecture -7-

Semantic, Intermediate Code Generation & Code Optimization

أ.م.د. سهاد مال الله

أ.م.د. عبير طارق مولود

أ.م. علاء نوري

تصميم المترجمات/الكورس الثاني/كل الافرع

قسم علوم الحاسوب/الجامعة التكنولوجية

2020-2021

Semantic Analysis

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

It uses the parse tree to identify the operators and operands of expressions and statements.

An important component is type checking.

The compiler checks that each operator has operands that are permitted by the source language specification.

Semantic Analysis

Static semantic checks are performed at compile time

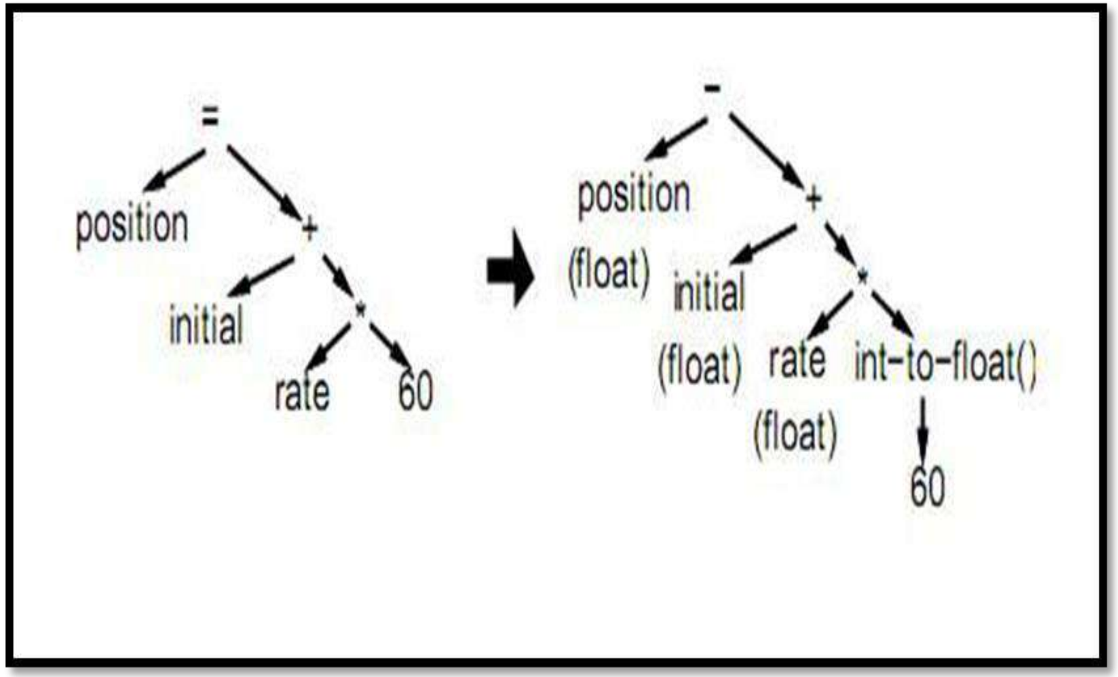
- **Type checking**
- **Every variable is declared before used**
- **Identifiers are used in appropriate contexts**
- **Check subroutine call arguments**

Dynamic semantic check are performed at run time, and the compiler produces code that performs these checks

- **Array subscript values are within bounds**
- **Arithmetic errors, e.g. division by zero**
- **A variable is used but hasn't been initialized**
- **When a check fails at run time, an exception is raised**

Type checking

A type checker verifies that the type of a construct matches that expected by its context. For example, the –in arithmetic operator mod in pascal requires integer operands, so a type checker must verify that the operands of mod have type integer.



Intermediate Code Generation

Translate from abstract-syntax trees to intermediate codes.

Generating a low-level intermediate representation with two properties:

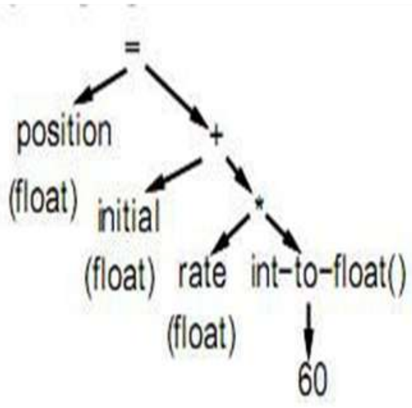
- ❖ It should be easy to produce
- ❖ It should be easy to translate into the target machine

One of the popular intermediate code is ***three-address code***. A three-address code:

Each statement contains at most 3 operands; in addition to ": =", i.e., assignment, at most one operator.

An" easy" and "universal" format that can be translated into most assembly languages.

Example:




```
temp1 := int-to-float(60)
temp2 := rate * temp1
temp3 := initial + temp2
position := temp3
```

Intermediate Code Generation

- Some of the basic operations which in the source program, to change in the Assembly language:

operations	H.L.L	Assembly language
Math. OP	+ , - , * , /	Add, sub, mult, div
Boolean. OP	&, | , ~	And, or, not
Assignment	=	mov
Jump	goto	JP, JN, JC
conditional	If, then	CMP
Loop instruction	For, do, repeat until, while do	These most have and I.C.G before change it to assembly language.

- The operation which change H.L.L to assembly language, is called the intermediate code generation and there is the division operation come with it, which mean every statement have a single operation.

Examples:

Ex (1):

$$X = \underbrace{\underbrace{A + \underbrace{\underbrace{B * C}_{T1} / D}_{T2}}_{T4} - \underbrace{Y * N}_{T3}}_{T5}$$

T1 = B * C
T2 = T1 / D
T3 = Y * N
T4 = A + T2
T5 = T4 - T3

Ex (2):

$$Y = \underbrace{\underbrace{\underbrace{\cos\ (\underbrace{A * B}_{T1})}_{T2} + \underbrace{C / N}_{T4}}_{T5} - \underbrace{Y * P}_{T3}}_{T6}$$

T1 = A * B
T2 = cos T1
T3 = Y * P
T4 = C / N
T5 = T2 + T4
T6 = T5 – T3

Mathematical operation

There are two kinds of operation, which are deals with mathematical operation, such as the parsing for these operations:

1. Triple form

Ex: X = A + B * C / (- N)

	OP	Arg1	Arg2
(0)	*	B	C
(1)	-	N	__
(2)	/	(0)	(1)
(3)	+	A	(2)
	=	X	(3)

Ex: Y = A + C * X / B [i]

	OP	Arg1	Arg2
(0)	*	C	X
(1)	=[ ]	B	i
(2)	/	(0)	(1)
(3)	+	A	(2)
	=	Y	(3)

Ex: X[i] = N * C / Y[i]

	OP	Arg1	Arg2
(0)	*	N	C
(1)	=[ ]	Y	i
(2)	/	(0)	(1)
(3)	[ ]=	X	i
	=	(3)	(2)

Ex: X = A + B * (c / d) - y

	OP	Arg1	Arg2
(0)	/	c	d
(1)	*	B	(0)
(2)	+	A	(1)
(3)	-	(2)	y
	=	X	(3)

Ex: A = C * X [i,j]

	OP	Arg1	Arg2
(0)	=[ ]	X	P
(1)	*	C	(0)
	=	A	(1)

2. Quadruple

form Ex: X = A * C / N + P

OP	Arg1	Arg2	Result
*	A	C	t1
/	t1	N	t2
+	t2	P	t3
=	t3		X

Ex: A = N[i] * C / N

OP	Arg1	Arg2	Result
=[ ]	N	i	t1
*	t1	C	t2
/	t2	N	t3
=	t3		A

Ex: A = C * y / X[i,j]

OP	Arg1	Arg2	Result
*	C	y	t1
=[ ]	X	P	t2
/	t1	t2	t3
=	t3		A

Ex: X = A + B * (c / d) - y

OP	Arg1	Arg2	Result
/	c	d	t1
*	B	t1	t2
+	A	t2	t3
-	t3	y	t4
=	t4		X

Ex: X[i] = a * c + y[i] – n[j] / V

OP	Arg1	Arg2	Result
*	a	c	t1
=[ ]	n	j	t2
/	t2	V	t3
=[ ]	y	i	t4
+	t1	t4	t5
-	t5	t3	t6
[ ]=	X	i	t7
=	t6		t7

Code Optimization

Compilers should produce target code that is as good as can be written by hand. The code produced by straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called Optimizations.

Function- Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common sub expression elimination, copy propagation, dead- code elimination, and constant floding are common examples of such function- preserving transformations.

الهدف :

1. تقليل المساحة الخزنية
2. زيادة سرعة التنفيذ
3. كلاهما

1. Common sub expressions

Ex1: $X = A + C * N - M$

$Y = B + C * N * e$

Sol: $Q = C * N$

$X = A + Q - M$

$Y = B + Q * e$

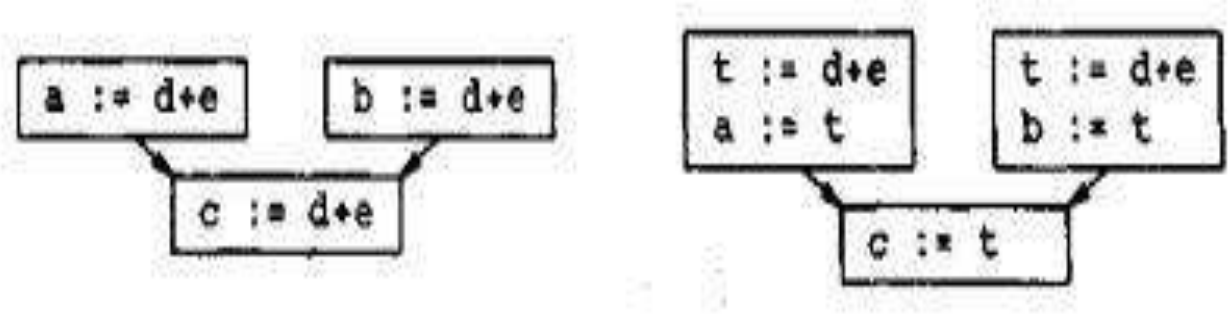
Ex2:

Before optimize	after optimize
t6 = 4 * i	t6 = 4 * i
X = a [t6]	X = a [t6]
t7 = 4 * i	t8 = 4 * j
t8 = 4 * j	t9 = a [t8]
t9 = a [t8]	a [t6] = t9
a [t7] = t9	a[t8] = X
t10 = 4 * j	
a[t10] = X	

Note: **The value of the variable which are optimize will not be change.**

Copy Propagation

Ex:



When the common sub expression in c = d + e is eliminated in previous section, the algorithm uses a new variable t to hold the value of d + e. Since control may reach c = d + e either after the assignment to a or after the assignment to b, it would be incorrect to replace c = d + e by either c=a or by c=b

Dead-Code Elimination

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used.

Ex1: X=3
If X > 4 Then
.
.
end

*This condition will not do, so we must use the optimization.

Ex2: A= false
If A Then
Begin
.
.
end

*This condition also will not do

Loop Optimization

The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside the loop. Three techniques are important for loop optimization: Code Motion, which moves code outside a loop; Induction Variable elimination; and, reduction in strength.

1- Code Motion

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop.

Ex1: While (I <= limit-2)

Sol: t = limit-2
While (I <= t)

Ex2: While X<A + C*y do
Begin
.
.
end

Sol: P = C * y
While X<A + P do
Begin
.
.
end

2- Induction Variable

Ex: $X = 2 * y + 2 * h + 4$

Sol: $X = 2 * (y + h) + 4$

Note: use the algebra method to optimize the mathematical expression.

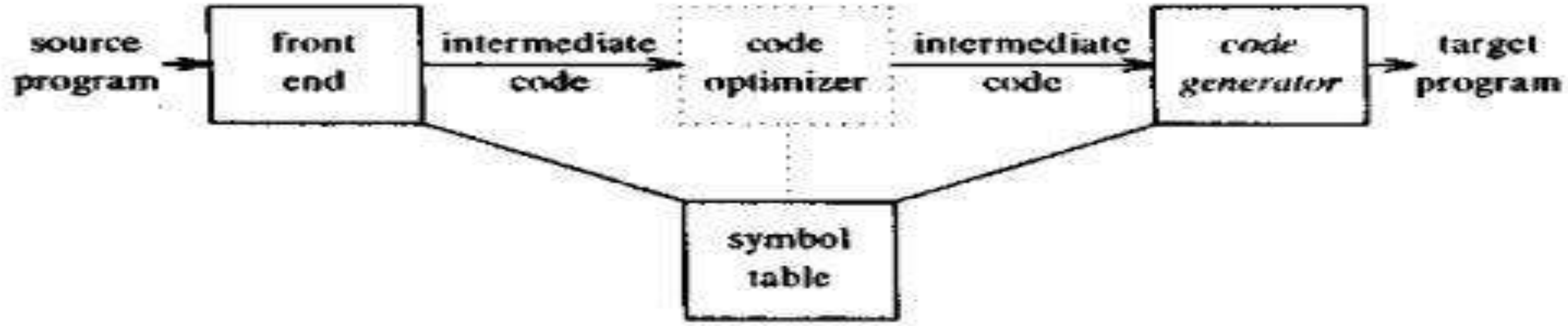
3- Reduction in Strength

Reduction in strength, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

Ex: $t4 = 4*j - 4$

Code Generation

The final phase in compiler is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program, as indicated in Fig



Position of code generator

Code generation takes a linear sequence of 3-address intermediate code instructions, and translates each instruction into one or more instructions. The big issues in code generation are

- Instruction selection
- Register allocation and assignment

Instruction selection:

for each type of three-address statement, we can design a code selection that outlines the target code to be generated for that construct.

Register allocation and assignment

The efficient utilization of registers involving operands is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1-Register allocation: selecting the set of variables that will reside in registers at each point in the program

2-Register assignment: selecting specific register that a variable reside in

The goal of these operations is generally to minimize the total number of memory accesses required by the program.

Thank you