University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب/ كل الافرع

Compiler Design
تصميم المترجمات

اعداد

ا.د. عبير طارق مولود
ا. علاء نوري مزهر

2025-2024

cs.uotechnology.edu.iq

Compiler Design

References:

1. Principle of compiler design
Alfred V. Aho & Jeffrey D. Ullman
2. Basics of compiler Design
Torben Egidius Mogensen
3. Compilers : principles, techniques, and
tools Alfred V. Aho & Jeffrey D. Ullman

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.**

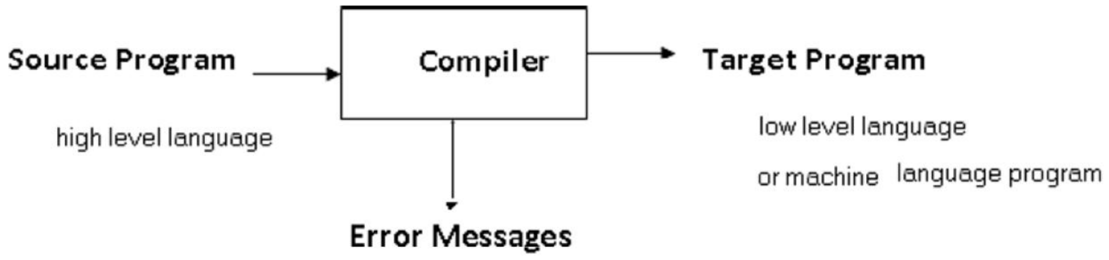


Figure (1) compiler structure

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).Figure (2) illustrate the compilation process Note that the program and data are processed at different times, compile time and run time respectively.

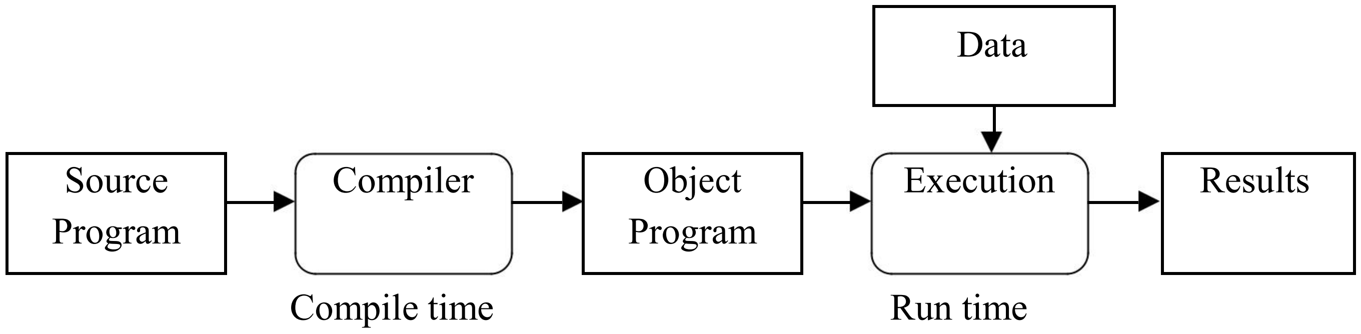


Figure (2) Compilation process

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 (3).

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, it should be easy to produce and 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.
We required several capabilities of the symbol table we need to be able to: 1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.
insert(s,t) : this function is to add a new name to the table
Lookup(s) : returns index of the entry for string s, or 0 if s is not found. 2- Access the information associated with a given name, and add new information for a given name.
3- Delete a name or group of names from the tables.
For example consider tokens **begin ,** we can initialize the symbol-table using the function**: insert("begin",1)**

-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.

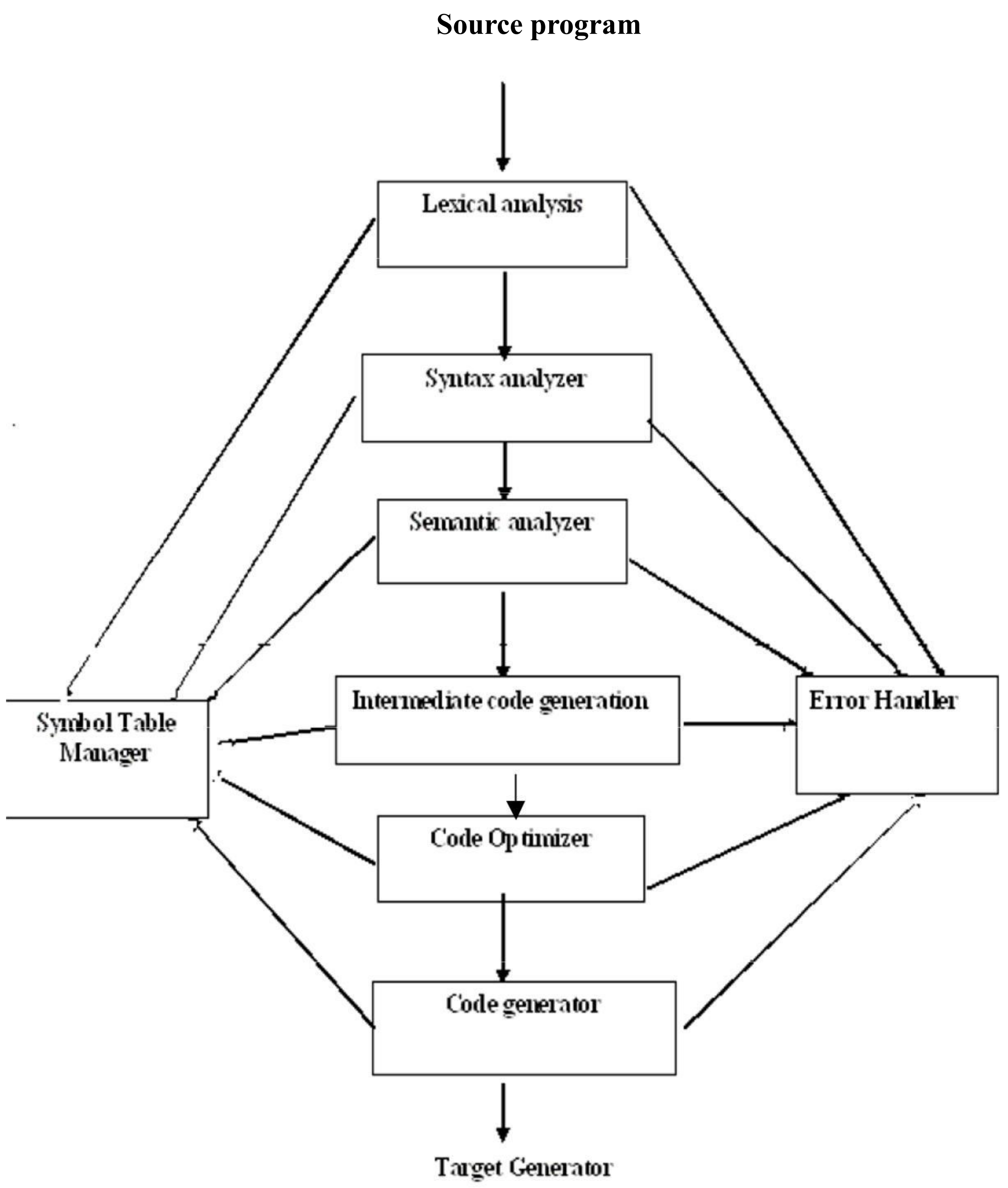


Figure (3) Phases of a Compiler

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.

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.. for syntax analyzer. This interaction, summarized in fig.6, is commonly implemented by making the lexical analyzer be a subroutine of the parser. Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

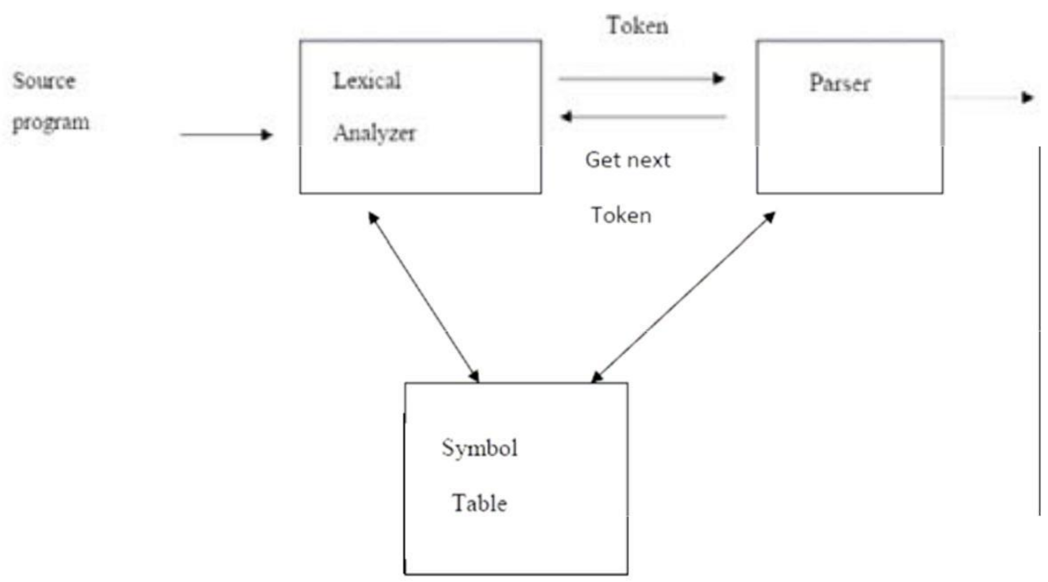


Figure (4) Interaction of lexical analyzer with parser

Preliminary scanning :

Since the lexical analyzer is the part of compiler that reads the source text; it may also perform certain secondary tasks at the user interface. One such task is **stripping out** from the source program comments and white space in the form of blank, tab, and new line characters. Another is **correlating error** messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of new line characters seen, so that a line number can be associated with an error message.

Some times, 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. For example, a FORTRAN compiler might use a scanner to eliminate blanks from the input.

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.

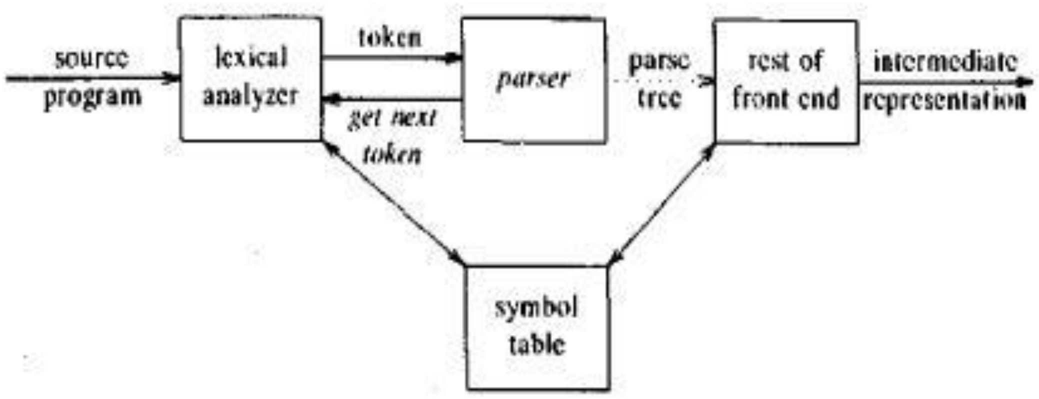


Figure (5) Position of parser in Compiler model

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.
We assume the output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

Top down parser

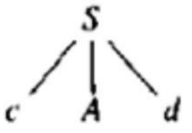
In this section there are basic ideas behind top-down parsing and show how constructs an efficient non- backtracking form of top-down parser called a predictive parser.

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.

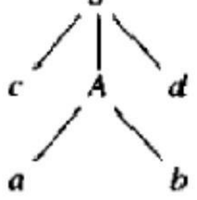
The following grammar requires **backtracking**:

$$
\begin{aligned}
S &\rightarrow cAd \\
A &\rightarrow ab \mid a
\end{aligned}
$$

the input string $w = cad$. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w. We then use the first production for S to expand the tree and obtain

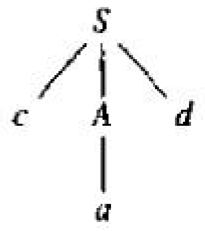


The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree



We now have a match for the second input symbol so we advance the input pointer to *d*, the third input symbol, and compare *d* against the next leaf, labeled *b*. Since *b* does not match *d*, we report failure and go back to *A* to see whether there is another alternative for *A* that we have not tried but that might produce a match.

now try the second alternative for A to obtain the tree:



The leaf *a* matches the second symbol of *w* and the leaf *d* matches the third symbol. Since we have produced a parse tree for *w*, we halt and announce successful completion of parsing.

- <u>problems of grammar</u>

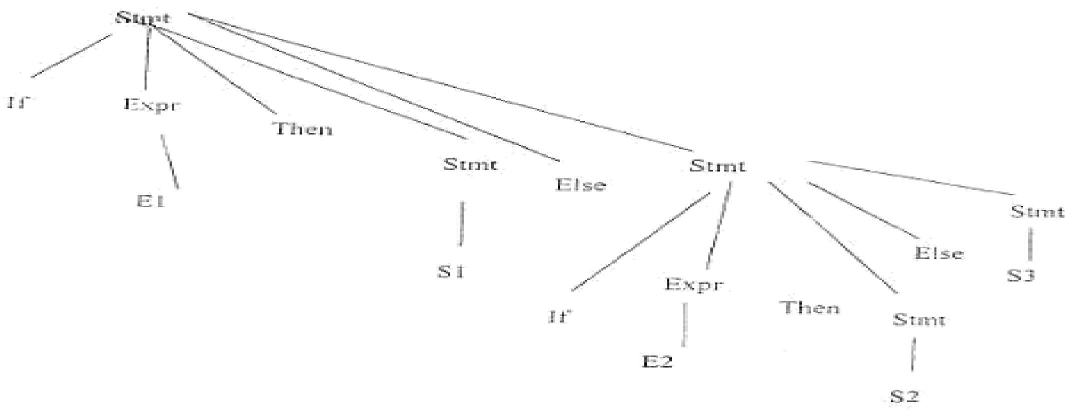
1- Ambiguity:

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.

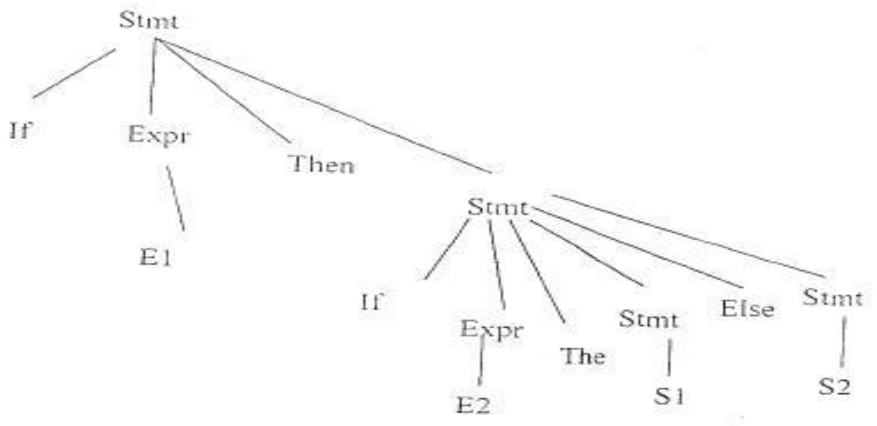
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. **As an example** ambiguous "else" grammar

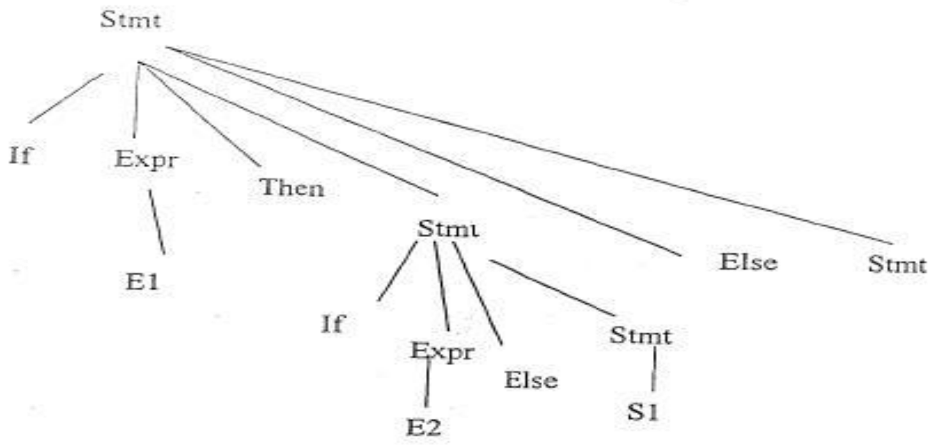
Stmt $\longrightarrow$ Expr then Stmt
| if Expr then Stmt else Stmt
|other

According to this grammar, the compound conditional statement
If El then S1 else if E2 then S2 else S3 has the parse tree link below:



The grammar above is ambiguous since the
string If E1 then if E2 then S1 else S2
Has the two parse trees shown below





2- Left Recursion

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

In the following example, we show how that left recursion pair of production could be replaced by the non-left recursion productions:

$A \longrightarrow A\alpha|\beta$

$A \longrightarrow A\alpha|\beta$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A'|\ \lambda$

Example: Consider the following grammar for arithmetic expressions.

$E \longrightarrow E+T\ |T$

$T \longrightarrow T*F|F$

$F \longrightarrow (E)\ |\ id$

Eliminating the immediate left recursion (productions of the form $A \longrightarrow A\alpha$ to the production for E and then for T, we obtain:

$E \longrightarrow TE'$

$E' \longrightarrow +T\ E'|\ \lambda$

$T \longrightarrow FT'$

$T' \longrightarrow *FT'\ |\ \lambda$

$F \longrightarrow (E)\ |\ id$

No matter how many A productions there are, we can eliminate immediate left recursion from them by the following technique. First we group the A production as

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

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

$A \longrightarrow \beta1A'|\ \beta2A'|\ \ldots|\ \beta nA'$

$A \longrightarrow \alpha1A'|\ \alpha2A'|\ \ldots|\ \alpha nA'\ |\ \lambda$

This produce eliminates all immediate left recursion from A and A' production. but it does not eliminate left recursion involving derivation of two or more steps.

$S \longrightarrow Aa \mid b$
$A \longrightarrow Ac \mid Sd \mid \lambda$
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

Stmt $\longrightarrow$ if Expr then Stmt else Stmt |
if Expr then Stmt

on seeing the input token if, we cannot immediately tell which production to choose to expand stmt. In general, if $A \longrightarrow \alpha\ \beta1 \mid \alpha\ \beta2$ are two A productions, and the input begins with a non -empty string derived from α ,we do not know whether to expand A to α β1 or to α β2. However, we may defer the decision by expanding A to αA', then after seeing the input derived from α, we expand A' to β1 or to β2. that is, left factored, the original production become:

$A \longrightarrow \alpha A'$
$A' \longrightarrow \beta1 \mid \beta2$

FIRST and FOLLOW:

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.

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 → **ε** ; is a production, then add **ε** 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 Y_i can derive epsilon **ε**; i++)
b- add First(Y_{i+1}) to First(X)
If Y1 does not derive **ε** , then we add nothing more to FIRST(X), but if

Y1→ **ε** , then we add FIRST(Y2) and so on .

First function example

1- FIRST (terminal) = {terminal }

$S \rightarrow aSb \mid ba \mid \varepsilon$

FIRST (a) ={a}

FIRST (b) ={b}

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

FIRST (S)= {a,b, **ε** }

To compute 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 → **ε**), then everything in FOLLOW(X) is in FOLLOW(A).

   i.e. : FOLLOW(A)= FOLLOW(X)

Follow function examples:

Example 1:

S → aSb │X

X → cXb │b

X → bXZ

$Z \rightarrow$ **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 | c**

$Y \rightarrow$ **b | ε**

	First	**Follow**
S	b	$
X	b , c	b , $
Y	b , ε	$

Example 3:

$S \rightarrow ABb \mid bc$

$A \rightarrow \varepsilon \mid abAB$

$B \rightarrow bc \mid cBS$

	First	Follow
S	b , a , c	$, b , c , a
A	ε , a	b , c

B	b , c	b , c , a

Example 4:

X → ABC | nX

A → bA | bb | ε

B → bA | CA

C → ccC | CA | cc

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

H.W:

S → bSX | Y

X → XC | bb

Y → b | bY

C → ccC | CX | cc

Note: there is a left recursion problem here trying to solve this Problem and find the first and follow for this grammar.

S → bSX | Y

X → bbX' **X' → CX'**| ε

Y → b | bY **C → ccC | CX | cc**

	First	Follow
S	b	$, b
X	b	$, b , c
X’	c , ε	$, b , c
Y	b	$, b
C	c	$, b , c

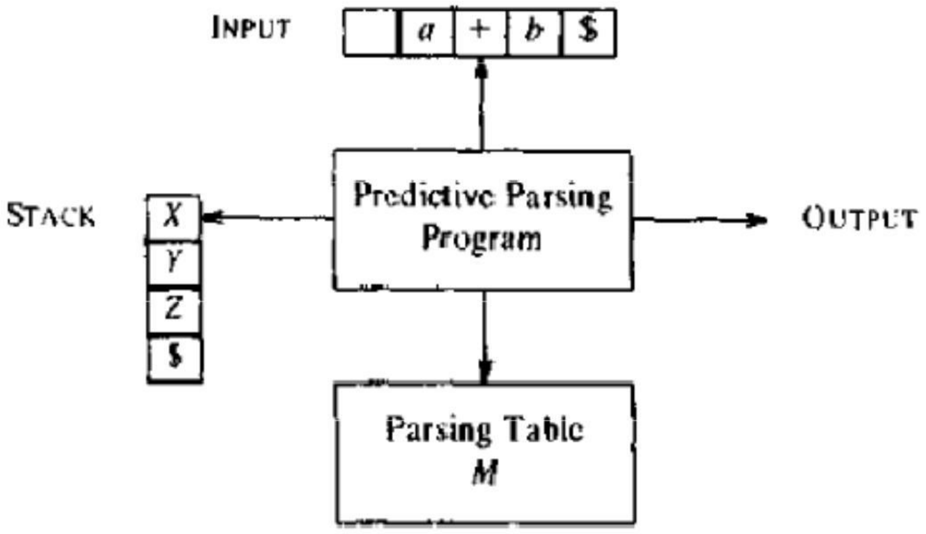
Predictive Parsing Method

In many cases, by carefully writing a grammar eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a non backtracking predictive parser.

We can build a predictive parser by maintaining a stack. The key problem during predictive parser is that of determining the production to be applied for a nonterminal. The nonrecursive parser looks up the production to be applied in a parsing table.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by \$, (a symbol used as a right endmarker to indicate the end of the input string). The stack contains a sequence of grammar symbols with \$ on the bottom,(indicating the bottom of the stack). Initially, the stack contains the start symbol of the grammar on

the top of \$. The parsing table is a two-dimensional array M[A,a], where A is a nonterminal, and **a** is a terminal or the symbol \$.



The parser is controlled by a program that behaves as follows. The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry $M[X, a]$ of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU (with U on top).

If M[X,a]= error, the parser calls an error recovery routine.

Example:

E→ E+T | T

T → T*F | F

F → (E) |id

Parse the input **id * id + id** by using predictive parsing:

1- we must solve the left recursion and left factoring if it founded in the grammar

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

2- we must find the first and follow to the grammar:

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

3-The parse table M for the grammar:

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

4-The moves made by predictive parser on input id+id*id

STACK	INPUT	OUTPUT
$\$E$	**id** + **id** * **id**$	
$\$E'T$	**id** + **id** * **id**$	$E \to TE'$
$\$E'T'F$	**id** + **id** * **id**$	$T \to FT'$
$\$E'T'$**id**	**id** + **id** * **id**$	$F \to$ **id**
$\$E'T'$	+ **id** * **id**$	
$\$E'$	+ **id** * **id**$	$T' \to \epsilon$
$\$E'T+$	+ **id** * **id**$	$E' \to +TE'$
$\$E'T$	**id** * **id**$	
$\$E'T'F$	**id** * **id**$	$T \to FT'$
$\$E'T'$**id**	**id** * **id**$	$F \to$ **id**
$\$E'T'$	* **id**$	
$\$E'T'F*$	* **id**$	$T' \to *FT'$
$\$E'T'F$	**id**$	
$\$E'T'$**id**	**id**$	$F \to$ **id**
$\$E'T'$	$	
$\$E'$	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

LL(1) grammars:

The previous algorithm can be applied to any grammar G to produce a parsing table M. For some grammars, M may have some entries that are multiply defined. If G is left recursive or ambiguous, then M will have at least one multiply-defined entry.

Example:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

FIRST(S) = { i, a}　　　　FOLLOW(S) = { \$, e }

FIRST(S’) = { e, ε}　　　　FOLLOW(S’) = { \$, e }

FIRST(E) = { b }　　　　FOLLOW(E) = { t }

So the parsing table for our grammar is:

NONTER-	INPUT SYMBOL					
MINAL	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 entry for M[S’,e] contains both S’→ eS and S’→ ε, since FOLLOW(S’) ={e, $}. The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (else) is seen. We can resolve the ambiguity if we choose S’→ eS. Note that the choice S’→ ε would prevent e from ever being put on the stack or removed from the input, and is therefore surely wrong.

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.

Bottom – Up Parsing

Bottom up parsers start from the sequence of terminal symbols and work their way back up to the start symbol by repeatedly replacing grammar rules' right hand sides by the corresponding non-terminal. This is the reverse of the derivation process, and is called "reduction".

Example: consider the grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

The sentence **abbcde** can be reduced to S by the following steps:

$abbcde$
$aAbcde$
$aAde$
$aABe$
S

Definition: a ***handle*** is a substring that

1- matches a right hand side of a production rule in the grammar and
2- Whose reduction to the nonterminal on the left hand side of that grammar rule is a step along the reverse of a rightmost derivation.

There is a general style of bottom-up syntax analysis, known as **shift reduces parsing.**

An easy to implement form of this parsing, called **operator precedence parsing**.

A much more general method of shift reduce parsing called **LR parsing** , used in a number of automatic parsing generators.

Shift reduces parsing attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (the top).

Shift Reduce Parsing Method

There are two problems that must be solved if we are to parse by handle pruning. The first is to determine the handel, and the second is to determine what production to choose in case there is more than one production with that substring on the right side. A convenient way to implement a shift reduce parser is to use stack to hold grammar symbols and an input buffer to hold the string (W) to be parsed. Use $ to mark the bottom of the stack and also the right end of the input. Initially the stack is empty and the string (W) is on the input, as follows:

Stack	**Input**
$	**W $**

The parser operates by shifting zero or more input symbol onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

Stack	**Input**
$ S	**$**

After entering this configuration the parser halts and announces successful completion of parsing.

At each step, the parser performs one of the following actions.

1- Shift one symbol from the input onto the parse stack
2- Reduce one handle on the top of the parse stack. The symbols from the right hand side of a grammar rule are popped of the stack, and the nonterminal symbol is pushed on the stack in their place.
3- Accept is the operation performed when the start symbol is alone on the parse stack and the input is empty.
4- Error actions occur when no successful parse is possible.

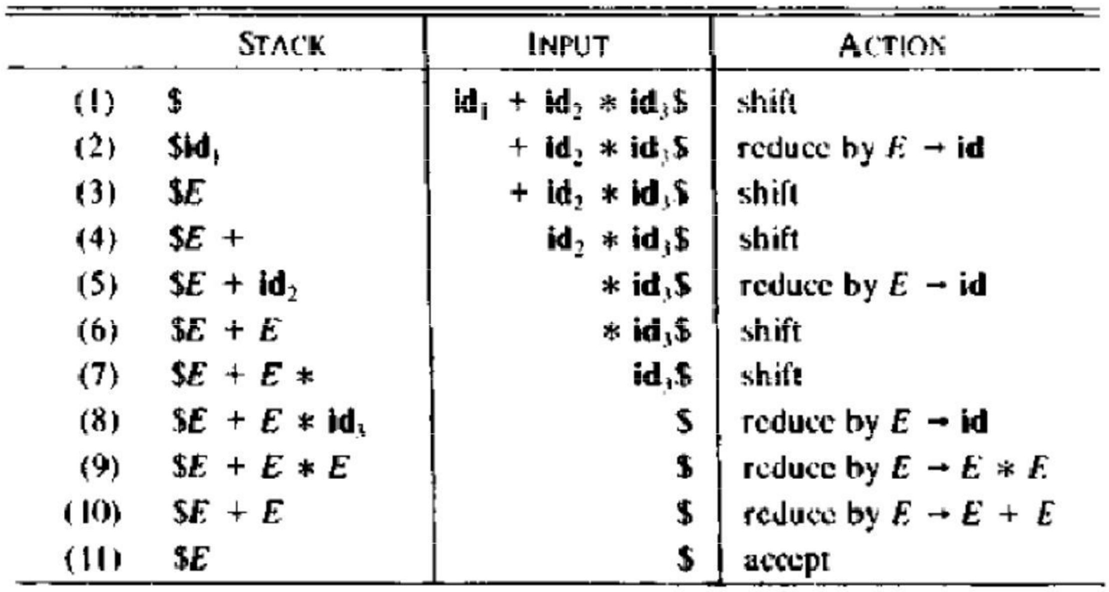
Example 1: parse the input id +id *id for this grammar

E → E+E

E → E*E

E → (E)

E → id



	STACK	INPUT	ACTION
(1)	$\$$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(2)	$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	reduce by $E \to \mathbf{id}$
(3)	$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(4)	$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	shift
(5)	$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	reduce by $E \to \mathbf{id}$
(6)	$\$E + E$	$* \mathbf{id}_3\$$	shift
(7)	$\$E + E *$	$\mathbf{id}_3\$$	shift
(8)	$\$E + E * \mathbf{id}_3$	$\$$	reduce by $E \to \mathbf{id}$
(9)	$\$E + E * E$	$\$$	reduce by $E \to E * E$
(10)	$\$E + E$	$\$$	reduce by $E \to E + E$
(11)	$\$E$	$\$$	accept

Example 2: parse the input id +* id for the same grammar

Stack	Input	Action
$	**id1 + * id2 $**	**Shift**
$ id1	**+ * id2 $**	**Reduce by E → id**
$ E+	*** id2 $**	**Shift**
$ E+*	**id2 $**	**Shift**
$ E +* id	**$**	**Shift**
$ E +*E	**$**	**Reduce by E→ id**
$ E +*E	**$**	**Not Accept**

H.W. : For this grammar

E → E+T | T

T → T*F | F

F → id | (E)

Parse the input **id * id + id**

1- Operator - precedence parser:

The operator-precedence parser is a shift –reduce parser that can be easily constructed by hand. It used for a small class of grammars which is called operator grammar. These grammars have the property:

- That no production right side is ε
- And no production right side has two adjacent nonterminals.

Example: The following grammar for expressions:

E → EAE | (E) | –E |id

A → + | - | * | / | ^

Is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals. However, if we substitute for A each of its alternatives, we obtain the following operator grammar:

E → E+E | E-E | E*E | E/E | E^E| (E) | –E | id

In operator-precedence parser, we define three disjoint precedence relations, <. , =, and .>, between certain pairs of terminals. These precedence relations guide the selection of handlers. If a <. b, we say **a** "yields precedence" to **b**; if a .> b , **a** "takes precedence over" **b**.

Now suppose we remove the nonterminals from the string and place the correct relation, <. , = or .> between each pair of terminals and between the endmost terminal and the $'s marking the ends of the string. For example, suppose we initially have the right-sentential form **id + id * id** and precedence relations are those given in below

1

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Then the string with the precedence relations inserted is:

$ <· id ·> + <· id ·> * <· id ·> $

The handle can be found by the following process:

1- Scan the string from the left end until the first .> is encountered. In our example, this occurs between the first **id** and **+.**
2- Then scan backwards (to the left) over any = until a<. is encountered, we scan back to $
3- The handle contains everything to the left of the first .> and to the right of the <. encountered in step (2).

So our first handler is the first **id.** We then reduce id to E. At this point we have the sentential form **E+id *id**. After reducing the two remaining id's to E by the same steps, we obtain the right-sentential form **E+E*E**. consider now the string $+*$ obtained by deleting the nonterminals. Inserting the precedence relations, we get:

$ <· + <· * ·> $

These precedence relations indicate that, in E+E*E, the handle is E*E and then E+E.

Stack implementation of operator precedence parser:

Suppose we are given the expression **id+id** to parse we set up the stack and input as:

<u>Stack   Input</u>

$ <.   id + id $

We shift id giving:

$ <. id   .>   + id $

The handle **id** is reduced to F

$ E   <.   + id $

$ <. E +   <.   id $

$ E + <.id   .> $

$ <. E + E   .> $

Now we are told to reduce the handle E+E:

$ E $

h.w.:

try input **id * (id ↑ id) − id / id** with the following relations

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*****	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

3

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 left-right 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.

1.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 I.
3. Find parsing table.
4. Check the sentence (parse the input).

The CLOSURE operation

If I is a set of items for a grammar G then the set of items CLOSURE(I) is constructed from I by the rules:

1. Every item in I is in CLOSURE(I).
2. lf A $\longrightarrow$ α.Bβ is in CLOSURE(I) and B$\longrightarrow$ γ is a production, then add the item B$\longrightarrow$. γ to I, if it is not already there.

Example: consider the grammar:

$$\begin{aligned}
E' &\rightarrow E\\
E &\rightarrow E + T \mid T\\
T &\rightarrow T * F \mid F\\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}$$

If I is the set of one item $\{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) contains the items

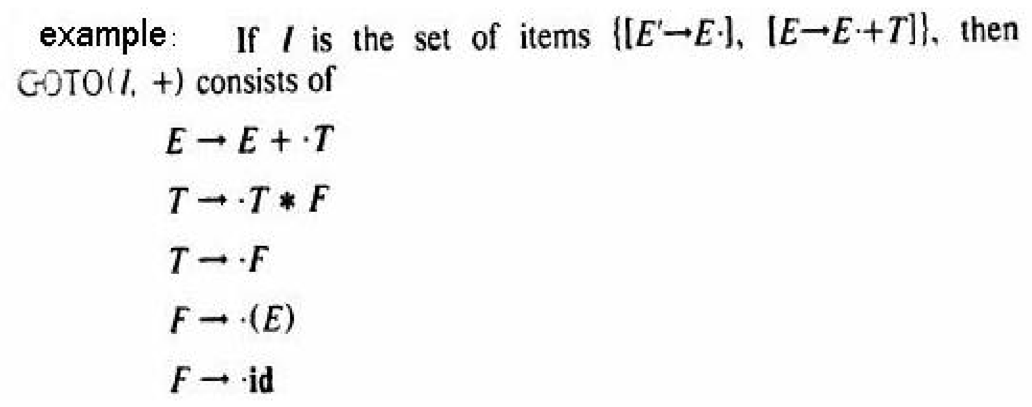
$$\begin{aligned}
E' &\rightarrow \cdot E\\
E &\rightarrow \cdot E+T\\
E &\rightarrow \cdot T\\
T &\rightarrow \cdot T*F\\
T &\rightarrow \cdot F\\
F &\rightarrow \cdot (E)\\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}$$

The function closure can be computed bellow:

```
function CLOSURE(I);
  begin
   J=I;
   repeat
     for each item $A \rightarrow \alpha.B\beta$ in I and each production $B \rightarrow \gamma$ in G such
     that $B \rightarrow .\gamma$ is not in I do
     add $B \rightarrow .\gamma$ to J.
   until no more items can be added to J;
 return J
end
```

GO TO operation

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $A \longrightarrow \alpha X.\ \beta$ such that $A \longrightarrow \alpha X.\ \beta$ is in I.


example: If I is the set of items $\{[E' \rightarrow E\cdot],\ [E \rightarrow E\cdot+T]\}$, then GOTO($I$, +) consists of

$$E \rightarrow E + \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot(E)$$
$$F \rightarrow \cdot\mathbf{id}$$

The Set of items Constructions:

```
  procedure ITEMS(G');
    begin
      c = {closure({$\grave{S} \longrightarrow .S$})};
     repeat
       for each set of items I in c and each grammar symbol x
       such that GOTO(I, x) is not empty and is not in c do
       add GOTO(I, x) to C
   until no more sets of items can be added to C
end
```

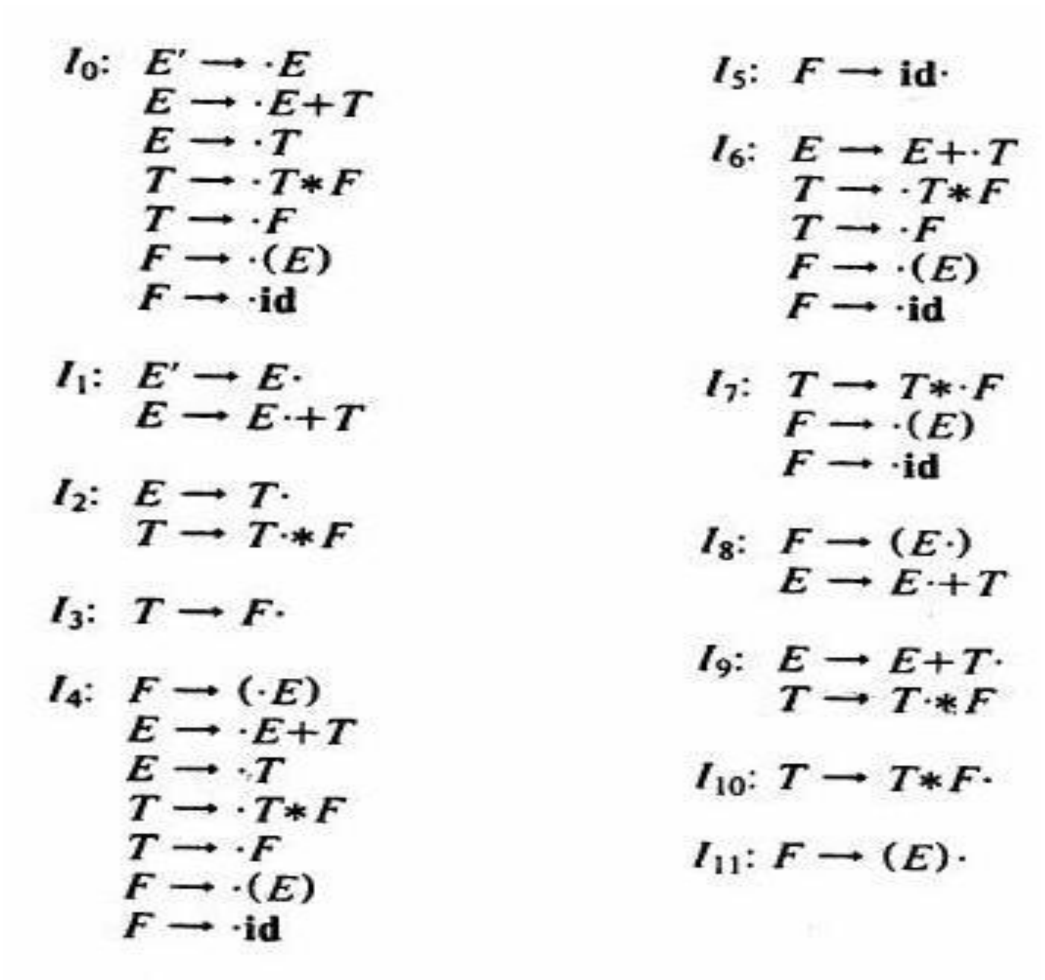
Example: consider the grammar:

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

1. Find first and follow

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

2. Find set of I.



I_0: $E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E\cdot+T$

I_2: $E \to T\cdot$
$T \to T\cdot*F$

I_3: $T \to F\cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id}\cdot$

I_6: $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_7: $T \to T*\cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_8: $F \to (E\cdot)$
$E \to E\cdot+T$

I_9: $E \to E+T\cdot$
$T \to T\cdot*F$

I_{10}: $T \to T*F\cdot$

I_{11}: $F \to (E)\cdot$

3. Find parsing table.

State	Action						Goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

4. Check the sentence (parse the input).

	Stack	Input
(1)	0	**id** * **id** + **id** $
(2)	0 **id** 5	* **id** + **id** $
(3)	0 *F* 3	* **id** + **id** $
(4)	0 *T* 2	* **id** + **id** $
(5)	0 *T* 2 * 7	**id** + **id** $
(6)	0 *T* 2 * 7 **id** 5	+ **id** $
(7)	0 *T* 2 * 7 *F* 10	+ **id** $
(8)	0 *T* 2	+ **id** $
(9)	0 *E* 1	+ **id** $
(10)	0 *E* 1 + 6	**id** $
(11)	0 *E* 1 + 6 **id** 5	$
(12)	0 *E* 1 + 6 *F* 3	$
(13)	0 *E* 1 + 6 *T* 9	$
(14)	0 *E* 1	$

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.

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

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

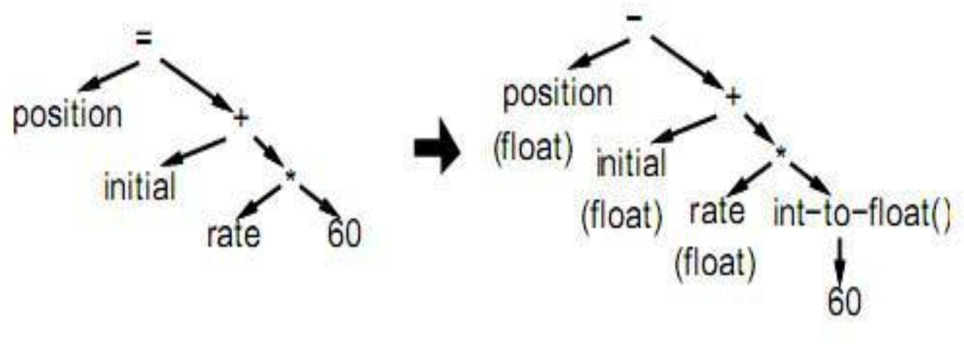
Arithmetic errors, e.g. division by zero

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:

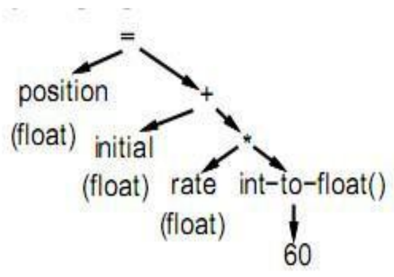
- It should be easy to produce
- It should be easy to translate into the target machine

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
```

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.

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.

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

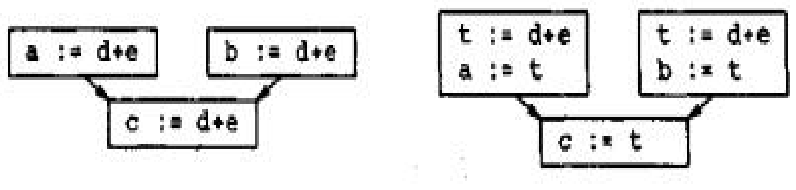
Ex2:

```
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.

2. 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

3. 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

4. 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.

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
```

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.

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

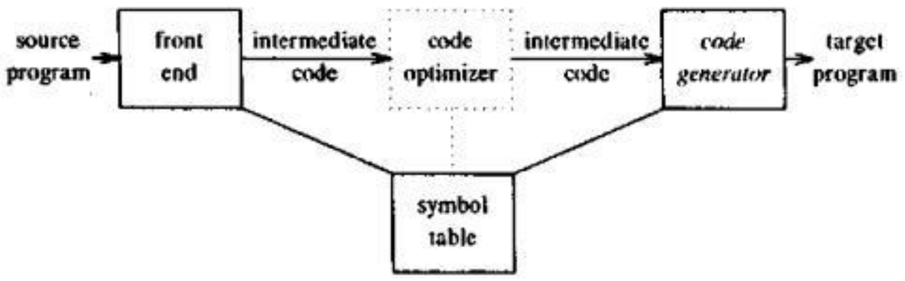


Figure (6) Position of code generator

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:

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

Resister 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.