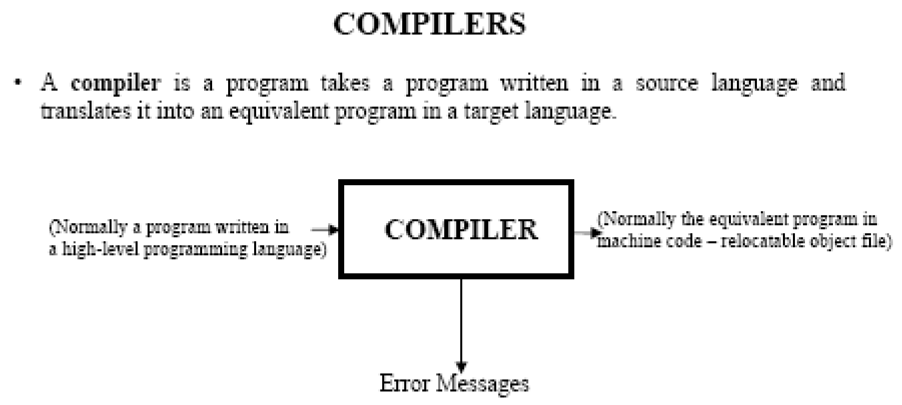
COMPUTER SCIENCE AND ENGINEERING
VI SEM CSE

Principles of Compiler Design Question and answers

---

1) What is a compiler?


COMPILERS
A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.
(Normally a program written in a high-level programming language)
COMPILER
(Normally the equivalent program in machine code – relocatable object file)
Error Messages

Simply stated, a compiler is a program that reads a program written in one language-the source language-and translates it into an equivalent program in another language-the target language (see fig.1) As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same.

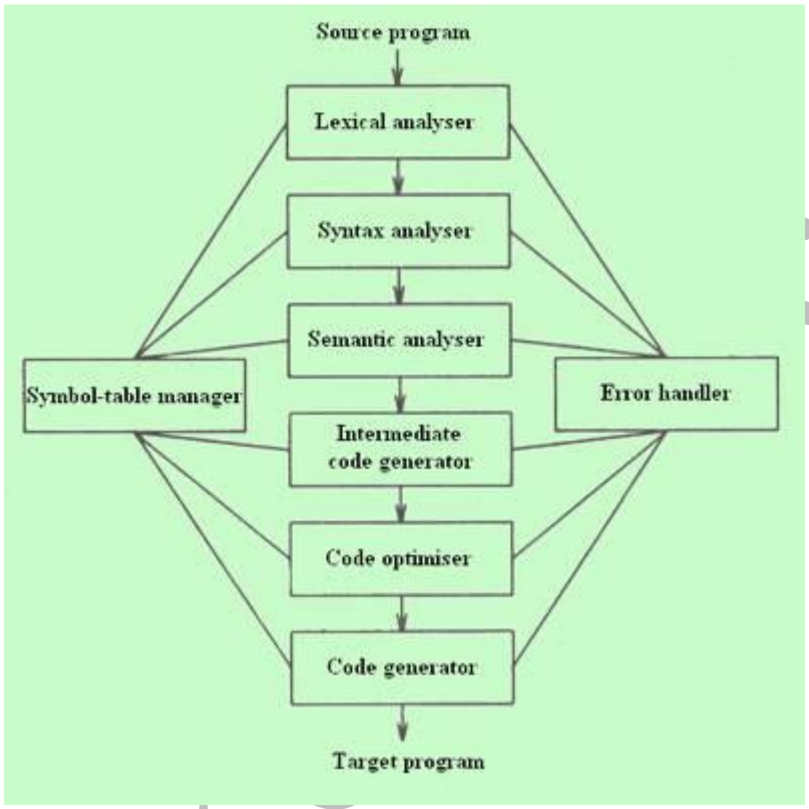
2) What are the phases of a compiler?
Phases of a Compiler

1. Lexical analysis ("scanning")
    - Reads in program, groups characters into "tokens"
2. Syntax analysis ("parsing")
    - Structures token sequence according to grammar rules of the language.
3. Semantic analysis
    - Checks semantic constraints of the language.
4. Intermediate code generation
    - Translates to "lower level" representation.
5. Program analysis and code optimization
    - Improves code quality.
6. Final code generation.

3) Explain in detail different phases of a compiler.

<u>THE DIFFERENT PHASES OF A COMPILER</u>

Conceptually, a compiler operates in *phases*, each of which transforms the source program from one representation to another.



The first three phases, forms the bulk of the analysis portion of a compiler. Symbol table management and error handling, are shown interacting with the six phases.

Symbol table management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lex analyzer, the identifier is entered into the symbol table.

Error Detection and Reporting

Each phase can encounter errors. A compiler that stops when it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors when the token stream violates the syntax of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

The Analysis phases

As translation progresses, the compiler's internal representation of the source program changes. Consider the statement,

position := initial + rate * 10

The lexical analysis phase reads the characters in the source pgm and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword etc. The character sequence forming a token is called the *lexeme* for the token. Certain tokens will be augmented by a 'lexical value'. For example, for any identifier the lex analyzer generates not only the token id but also enter s the lexeme into the symbol table, if it is not already present there. The lexical value associated this occurrence of id points to the symbol table entry for this lexeme. The representation of the statement given above after the lexical analysis would be:

id1: = id2 + id3 * 10

Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax trees (fig 3).

Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can have a variety of forms.

In three-address code, the source pgm might look like this,

```
temp1: = inttoreal (10)

temp2: = id3 * temp1

temp3: = id2 + temp2

id1: = temp3
```

Code Optimisation

The code optimization phase attempts to improve the intermediate code, so that faster running machine codes will result. Some optimizations are trivial. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called 'optimising compilers', a significant fraction of the time of the compiler is spent on this phase.

Code Generation

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

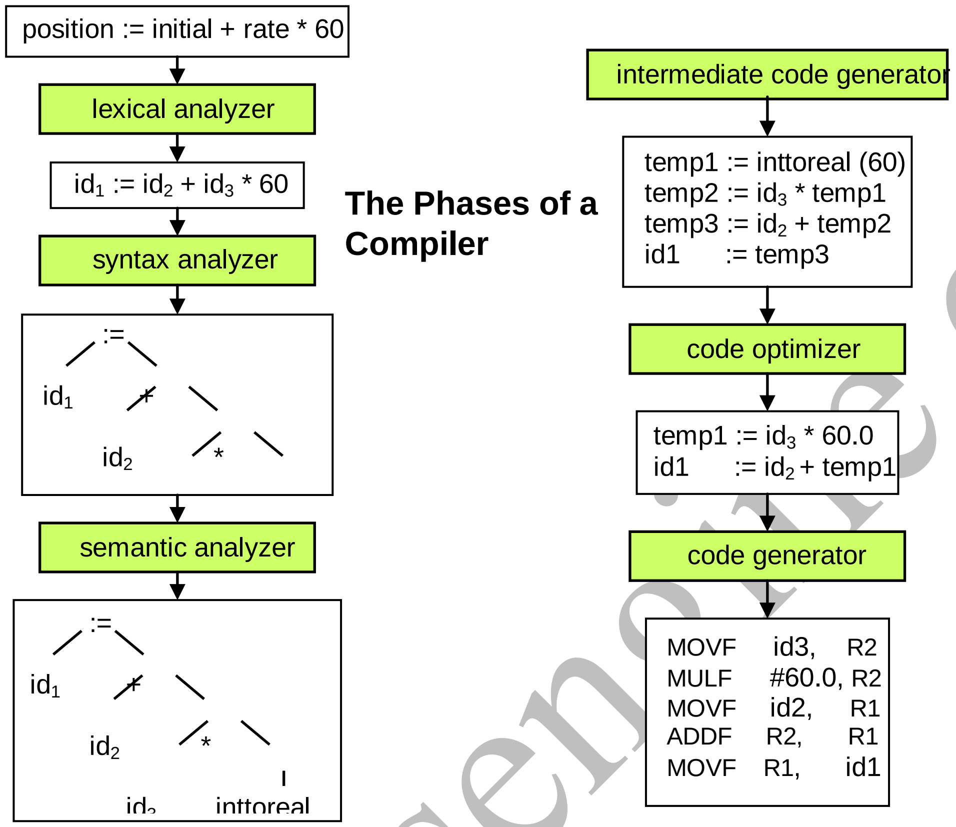
4) What is grouping of phases?

Grouping of Phases

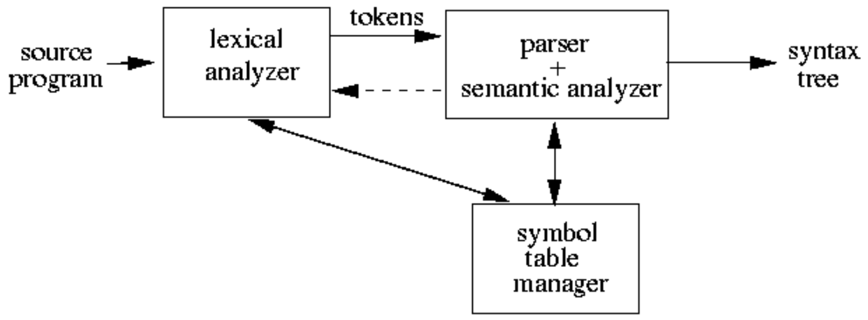
- *Front end* : machine independent phases
    - Lexical analysis
    - Syntax analysis
    - Semantic analysis
    - Intermediate code generation

    - Some code optimization
- *Back end* : machine dependent phases
    - Final code generation
    - Machine-dependent optimizations

5) Explain with diagram how a statement is compiled .



6)What are roles and tasks of a lexical analyzer?



Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.

Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

- **Define Context free grammar.**

- o **Context-free Grammars**
- o A *context-free grammar* for a language specifies the syntactic structure of programs in that language.
- o Components of a grammar:
  - o a finite set of tokens (obtained from the scanner);
  - o a set of variables representing "related" sets of strings, e.g., *declarations, statements,* expressions.
  - o a set of rules that show the structure of these strings.
  - o an indication of the "top-level" set of strings we care about.
- o **Context-free Grammars: Definition**
- o Formally, a context-free grammar *G* is a 4-tuple *G* = (V, T, P, S), where:
  - o V is a finite set of *variables* (or *nonterminals*). These describe sets of "related" strings.
  - o T is a finite set of *terminals* (i.e., tokens).
  - o P is a finite set of *productions*, each of the form
- o $A \rightarrow \alpha$
- o where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals.
  - o $S \in V$ is the *start symbol*.

Example of CFG :

E ==>EAE | (E) | -E | id

A==> + | - | * | / |

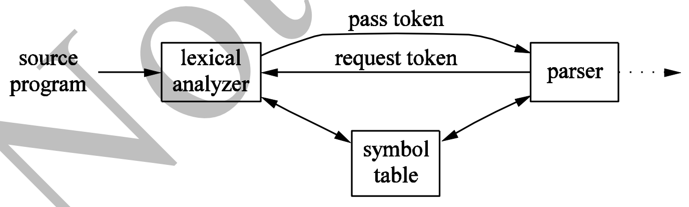
Where E,A are the non-terminals while id, +, *, -, /,(,) are the terminals.

6) What are parsers?

Parser

- Accepts string of tokens from lexical analyzer (usually one token at a time)
- Verifies whether or not string can be generated by grammar
- Reports syntax errors (recovers if possible)

THE ROLE OF A PARSER



Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion.

The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing.

7) What are parse trees?

Parse Trees

• Nodes are non-terminals.
• Leaves are terminals.
• Branching corresponds to rules of the grammar.
• The leaves give a sentence of the input language.
• For every sentence of the language there is at least one parse tree.
• Sometimes we have more then one parse tree for a sentence.
• Grammars which allow more than one parse tree for some sentences are called ambiguous and are usually not good for compilation.

8) What are different kinds of errors encountered during compilation?

Compiler Errors

- Lexical errors (e.g. misspelled word)
- Syntax errors (e.g. unbalanced parentheses, missing semicolon)
- Semantic errors (e.g. type errors)
- Logical errors (e.g. infinite recursion)

Error Handling

- Report errors clearly and accurately
- Recover quickly if possible
- Poor error recover may lead to avalanche of errors

9) What are different error recovery strategies?

Error Recovery strategies

- **Panic mode**: discard tokens one at a time until a synchronizing token is found
- **Phrase-level recovery**: Perform local correction that allows parsing to continue
- **Error Productions**: Augment grammar to handle predicted, common errors
- **Global Production**: Use a complex algorithm to compute least-cost sequence of changes leading to parseable code

10) Explain Recursive descent parsing.

Recursive descent parsing: corresponds to finding a leftmost derivation for an input string
Equivalent to constructing parse tree in pre-order
Example:
Grammar: S ! cAd A ! ab j a
Input: cad
Problems:
1. backtracking involved ()buffering of tokens required)
2. left recursion will lead to infinite looping
3. left factors may cause several backtracking steps
Compiler Construction: Parsing – p. 3/31

11) Give an example of ambiguous grammar.

Examples
Ambigous grammar:
E ::= E ”_” E | E ”+” E | ”1” | ”(” E ”)”
Unambigous grammar
E ::= E ”+” T | T
T ::= T ”_” F | F
F ::= ”1” | ”(” E ”)”

8) What is left recursion? How it is eliminated?

Left recursion: G istb left recursive if for some non-terminal A,
$A \overset{+}{\Rightarrow} A\alpha$

Simple case I: $A \to A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}$

Simple case II:

$$\begin{array}{lcl} A & \to & A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \\ & & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

$$\Downarrow$$

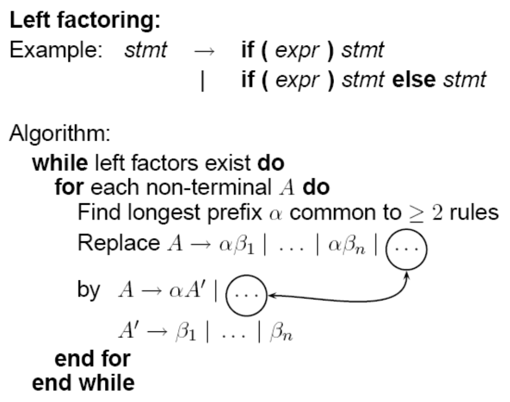
$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \to & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

12) What is left factoring?
Left Factoring
- Rewriting productions to delay decisions
- Helpful for predictive parsing
- Not guaranteed to remove ambiguity

$A \to \alpha\beta1 \mid \alpha\beta2$

$$\Downarrow$$

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



Left factoring:

Example: *stmt* → **if (** *expr* **)** *stmt*
　　　　　　| **if (** *expr* **)** *stmt* **else** *stmt*

Algorithm:

```
while left factors exist do
  for each non-terminal $A$ do
    Find longest prefix $\alpha$ common to $\geq 2$ rules
    Replace $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \boxed{\ldots}$
    by $A \rightarrow \alpha A' \mid \boxed{\ldots}$
       $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$
  end for
end while
```

13) What is top down parsing?

Top Down Parsing

- Can be viewed two ways:
  - Attempt to find leftmost derivation for input string
  - Attempt to create parse tree, starting from at root, creating nodes in preorder
- General form is recursive descent parsing
  - May require backtracking
  - Backtracking parsers not used frequently because not needed

14) What is predictive parsing?

- A special case of recursive-descent parsing that does not require backtracking
- Must always know which production to use based on current input symbol
- Can often create appropriate grammar:
  - removing left-recursion
  - left factoring the resulting grammar

15) Define LL(1) grammar.

LL(1) Grammars

- Algorithm covered in class can be applied to any grammar to produce a parsing table
- If parsing table has no multiply-defined entries, grammar is said to be "LL(1)"
  - First "L", left-to-right scanning of input
  - Second "L", produces leftmost derivation
  - "1" refers to the number of lookahead symbols needed to make decisions

16) What is shift reduce parsing?

Shift-Reduce Parsing

- One simple form of bottom-up parsing is shift-reduce parsing
- Starts at the bottom (leaves, terminals) and works its way up to the top (root, start symbol)
- Each step is a "reduction":
  - Substring of input matching the right side of a production is "reduced"
  - Replaced with the nonterminal on the left of the production
- If all substrings are chosen correctly, a rightmost derivation is traced in reverse

Shift-Reduce Parsing Example

```
S → aABe
A → Abc | b
B -> d
```

```
abbcde
aAbcde
aAde
aABe
S
```

$S \xrightarrow{}_{rm} aABe \xrightarrow{}_{rm} aAde \xrightarrow{}_{rm} aAbcde \xrightarrow{}_{rm} abbcde$

17) Define Handle.

Handles

- Informally, a "**handle**" of a string:
  - Is a substring of the string
  - Matches the right side of a production
  - Reduction to left side of production is one step along **reverse of rightmost derivation**
- Leftmost substring matching right side of production is not necessarily a handle
  - Might not be able to reduce resulting string to start symbol
  - In example from previous slide, if reduce aAbcde to aAAcde, can not reduce this to S
- **Formally, a handle of a right-sentential form γ:**
  - Is a production A ◊ β and a position of γ where β may be found and replaced with A
  - Replacing A by β leads to the previous right-sentential form in a rightmost derivation of γ
- So if S rm*=> αAw rm=> αβw then A ◊ β in the position following α is a handle of αβw
- The string w to the right of the handle contains only terminals
- Can be more than one handle if grammar is ambiguous (more than one rightmost derivation)

18) What is handle pruning?

- Repeat the following process, starting from string of tokens until obtain start symbol:
  - Locate handle in current right-sentential form
  - Replace handle with left side of appropriate production
- Two problems that need to be solved:
  - How to locate handle
  - How to choose appropriate production

19) Explain stack implementation of shift reduce parsing.

Shift-Reduce Parsing

- Data structures include a stack and an input buffer
  - Stack holds grammar symbols and starts off empty
  - Input buffer holds the string w to be parsed
- Parser shifts input symbols onto stack until a handle β is on top of the stack
  - Handle is reduced to the left side of appropriate production
  - If stack contains only start symbol and input is empty, this indicates success

Actions of a Shift-Reduce Parser

- **Shift** – the next input symbol is shifted onto the top of the stack
- **Reduce** – The parser reduces the handle at the top of the stack to a nonterminal (the left side of the appropriate production)
- **Accept** – The parser announces success
- **Error** – The parser discovers a syntax error and calls a recovery routine

Shift Reduce Parsing Example

Stack	Input	Action
$	id1 + id2 * id3$	shift
$id1	+ id2 * id3$	reduce by E → id
$E	+ id2 * id3$	shift
$E +	id2 * id3$	shift
$E + id2	* id3$	reduce by E → id
$E + E	* id3$	shift
$E + E *	id3$	shift
$E + E * id3	$	reduce by E → id
$E + E * E	$	reduce by E → E * E
$E + E	$	reduce by E → E + E
$E	$	accept

20) What are viable prefixes?

Viable Prefixes

- Two definitions of a viable prefix:
  - A prefix of a right sentential form that can appear on a stack during shift-reduce parsing
  - A prefix of a right-sentential form that does not continue past the right end of the rightmost handle
- Can always add tokens to the end of a viable prefix to obtain a right-sentential form

21) Explain conflicts in shift reduce parsing.

Conflicts in Shift-Reduce Parsing

- There are grammars for which shift-reduce parsing can not be used

- **Shift/reduce conflict**: can not decide whether to shift or reduce
- **Reduce/reduce conflict**: can not decide which of multiple possible reductions to make
- Sometimes can add rule to adapt for use with ambiguous grammar

22) What is operator precedence parsing?
Operator-Precedence Parsing

- A form of shift-reduce parsing that can apply to certain simple grammars
  - No productions can have right side ε
  - No right side can have two adjacent nonterminals
  - Other essential requirements must be met
- Once the parser is built (often by hand), the grammar can be effectively ignored

23) What are precedence relations?
Precedence Relations

Relation	**Meaning**
a <· b	**a "yields precedence to" b**
a ·= b	**a "has the same precedence as" b**
a ·> b	**a "takes precedence over" b**

24) How precedence relations are used?
Using Precedence Relations (1)

- Can be thought of as delimiting handles:
  - <· Marks left end of handle
  - ·> Appears in the interior of handle
  - ·= Marks right end of handle
- Consider right-sentential β0a1β1β1…anBn:
  - Each βi is either single nonterminal or ε
  - Each ai is a single token
  - Suppose that exactly one precedence relation will hold for each ai, ai+1 pair

Using Precedence Relations (2)

- Mark beginning and end of string with $
- Remove the nonterminals
- Insert correct precedence relation between each pair of terminals

	i		*	$
i		·	·	·
+	<	·	<	·
*	<	·	·	·
$	<	<	<	

```
id + id * id
```

⇓

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

Using Precedence Relations (3)

- To find the current handle:
  - Scan the string from the left until the first ·> is encountered
  - Scan backwards (left) from there until a <· is encountered
  - Everything in between, including intervening or surrounding nonterminals, is the handle
- The nonterminals do not influence the parse!

25) Explain operator precedence parsing algorithm.

```
set ip to point to the first symbol in w$
initialize stack to $
repeat forever
  if $ on top of stack in ip points to $
    return success
  else
    let a be topmost symbol on stack
    let b be symbol pointed to by ip
    if a <· b or a ·= b
      push b onto stack
      advance ip to next input symbol
    else if a ·> b
      repeat
        pop x
      until top symbol on stack <· x
    else
      error()
```

26) Explain a heuristic to produce a proper set of precedence relations.

Precedence and Associativity (1)

- For grammars describing arithmetic expressions:
  - Can construct table of operator-precedence relations automatically
  - Heuristic based on precedence and associativity of operators
- Selects proper handles, even if grammar is ambiguous

The following rules are designed to select the proper handles to reflect a given set of associativity and precedence rules for binary operators :

- If operator θ1 has higher precedence than operator θ2, make θ1 ·> θ2 and θ2 <· θ1

- If θ1 and θ2 are of equal precedence:
  - If they are left associative, make θ1 ·> θ2 and θ2 ·> θ1
  - If they are right associative, make θ1 <· θ2 and θ2 <· θ1

27) What is an operator grammar?

A grammar having the property (among other essential requirements) that no production right side is ε or has two adjacent nonterminals is called an **operator grammar**.

Operator Grammar Example

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

- ^ is of highest precedence and is right-associative
- * and / are of next highest precedence and are left-associative
- + and – are of lowest precedence and are left-associative

28) What are precedence functions?

Precedence Functions (1)

- Do not need to store entire table of precedence relations
- Select two precedence functions f and g :
  - f(a) < g(b) whenever a <· b
  - f(a) = g(b) whenever a ·= b
  - f(a) > g(b) whenever a ·> b

	+	-	*	/	^	(	)	i	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3		5	5	0	5	0

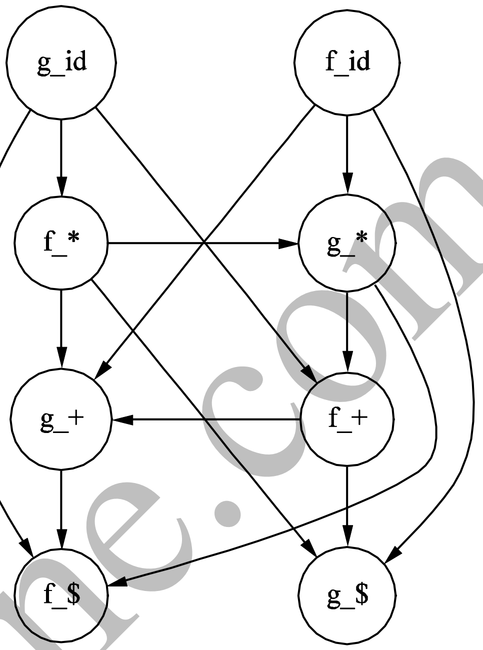
29) How precedence functions are constructed?

Precedence Functions Algorithm

- Create symbols fa and fg for all tokens and $
- If a ·= b then fa and gb must be in same group
- Partition symbols into as many groups as possible
- For all cases where a <· b, draw edge from group of gb to group of fa
- For all cases where a ·> b, draw edge from group of fa to group of gb
- If graph has cycles, no precedence functions exist
- Otherwise:
  - f(a) is the length of the longest path beginning at group of fa
  - g(a) is the length of the longest path beginning at group of ga

Precedence Functions Example

	i	+	*	$
i		·	·	·
+	<	·	<	·
*	<	·	·	·
$	<	<	<	



	+	*	i	$
f	2	4	4	0
g	1	3	5	0

30) How error recovery is enforced in operator precedence parsers?
Detecting and Handling Errors

- Errors can occur at two points:
  - If no precedence relation holds between the terminal on top of stack and current input
  - If a handle has been found, but no production is found with this handle as right side
- Errors during reductions can be handled with diagnostic message
- Errors due to lack of precedence relation can be handled by recovery routines specified in table

31) What are LR parsers?
LR Parsers

- LR Parsers us an efficient, bottom-up parsing technique useful for a large class of CFGs
- Too difficult to construct by hand, but automatic generators to create them exist (e.g. Yacc)
- LR(k) grammars
  - "L" refers to left-to-right scanning of input

  - "R" refers to rightmost derivation (produced in reverse order)
  - "k" refers to the number of lookahead symbols needed for decisions (if omitted, assumed to be 1)

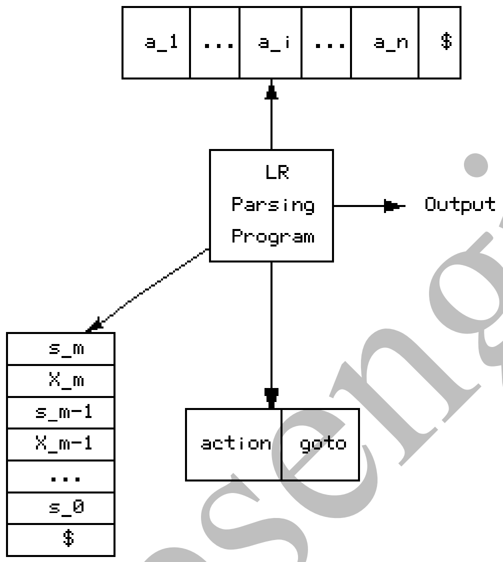
32) What are the benefits of LR parsers?

Benefits of LR Parsing

- Can be constructed to recognize virtually all programming language construct for which a CFG can be written
- Most general non-backtracking shift-reduce parsing method known
- Can be implemented efficiently
- Handles a class of grammars that is a superset of those handled by predictive parsing
- Can detect syntactic errors as soon as possible with a left-to-right scan of input

33) Explain LR parsing Algorithm with diagrams. For a given grammar and parsing table Tabulate the moves of LR parser for a given input string id * id + id.

Model of LR Parser



The LR parsing program works as follows :

The schematic of LR parser consists of an input,an output,a stack,a driver program,a parsing table that has two parts (actions and goto)

- Driver program is the same for all LR Parsers
- Stack consists of states (si) and grammar symbols (Xi)
  - Each state summarizes information contained in stack below it
  - Grammar symbols do not actually need to be stored on stack in most implementations
- State symbol on top of stack and next input symbol used to determine shift/reduce decision
- Parsing table includes action function and goto function
- **Action function**
  - Based on state and next input symbol
  - Actions are shift, reduce, accept or error
- **Goto function**
  - Based on state and grammar symbol
  - Produces next state
- Configuration (s0X1s1…Xm sm,ai ai+1…an$) indicates right-sentential form X1X2…Xmaiai+1…an

- If action[sm,ai] = shift s, enter configuration (s0X1s1…Xmsmais,ai+1…an$)
- If action[sm,ai] = reduce A ◊ B, enter configuration (s0X1s1…Xm-rsm-rAs, ai+1…an$), where s = goto[sm-r,A] and r is length of B
- If action[sm,ai] = accept, signal success
- If action[sm,ai] = error, try error recovery

LR Parsing Algorithm

```
set ip to point to the first symbol in w$
initialize stack to s0
repeat forever
  let s be topmost state on stack
  let a be symbol pointed to by ip
  if action[s,a] = shift s'
    push a then s' onto stack
    advance ip to next input symbol
  else if action[s,a] = reduce A → B
    pop 2*|B| symbols of stack
    let s' be state now on top of stack
    push A then goto[s',A] onto stack
    output production A → B
  else if action[s,a] == accept
    return success
  else
    error()
```

LR Parsing Table Example

```
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id
```

state	action						goto		
	i	+	*	(	)	$	E	T	F
0	s			s			1	2	3
1		s				a			
2		r	s		r	r			
3		r	r		r	r			
4	s			s			8	2	3
5		r	r		r	r			
6	s			s				9	3
7	s			s					1
8		s			s				

LR Parsing Example
Moves of LR parser on id * id + id

Stack	Input	Action
(1) s0	id * id + id $	shift
(2) s0 id s5	* id + id $	reduce by F → id
(3) s0 F s3	* id + id $	reduce by T → F
(4) s0 T s2	* id + id $	shift
(5) s0 T s2 * s7	id + id $	shift
(6) s0 T s2 * s7 id s5	+ id $	reduce by F → id
(7) s0 T s2 * s7 F s10	+ id $	reduce by T → T * F
(8) S0 T s2	+ id $	reduce by E → T
(9) s0 E s1	+ id $	shift
(10) s0 E s1 + s6	id $	shift
(11) s0 E s1 + s6 id s5	$	reduce by F → id
(12) s0 E s1 + s6 F s3	$	reduce by T → F
(13) s0 E s1 + s6 T s9	$	reduce by E → E + T
(14) s0 E s1	$	accept

34) What are three types of LR parsers?

Three methods:

a. **SLR (simple LR)**
   i. Not all that simple (but simpler than other two)!
   ii. Weakest of three methods, easiest to implement
b. **Constructing canonical LR parsing tables**
   i. Most general of methods
   ii. Constructed tables can be quite large
c. **LALR parsing table (lookahead LR)**
   i. Tables smaller than canonical LR
   ii. Most programming language constructs can be handled

35) Explain the non-recursive implementation of predictive parsers.

Non-recursive implementation:

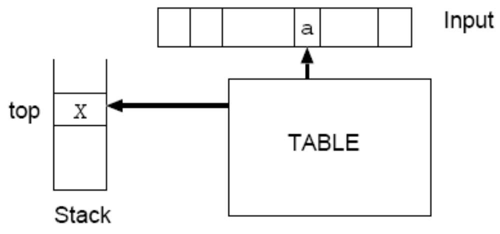


Table: 2-d array s.t. $M[A, a]$ specifies A-production to be used if input symbol is a

Algorithm:

0. Initially: stack contains $\langle$EOF $S\rangle$, input pointer is at start of input
1. if $X = a =$ EOF, done
2. if $X = a \neq$ EOF, pop stack and advance input pointer
3. if X is non-terminal, lookup $M[X, a] \quad \Rightarrow \quad X \rightarrow UVW$ pop X, push W, V, U

FIRST(α): set of terminals that begin strings derived from α

if $\alpha \overset{*}{\Rightarrow} \epsilon$, then $\epsilon \in FIRST(\alpha)$

FOLLOW (A): set of terminals that can appear immediately to the right of A in some sentential form

$$FOLLOW(A) = \{a \mid S \overset{*}{\Rightarrow} \alpha A a \beta\}$$

if A is the rightmost symbol in any sentential form, then EOF $\in FOLLOW(A)$

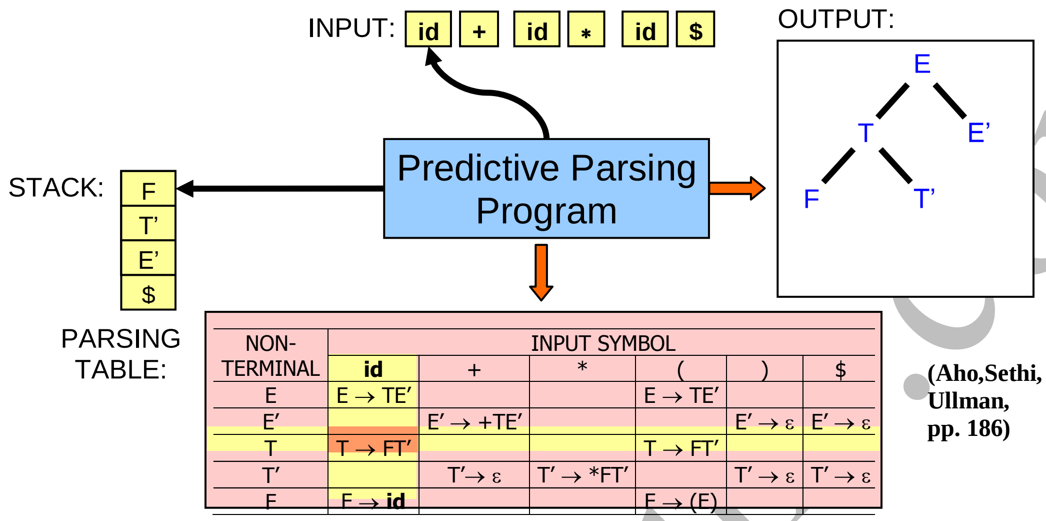
FIRST:

1. if X is a terminal, then $FIRST(X) = \{X\}$
2. if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$
3. if $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production:
   if $a \in FIRST(Y_i)$ and $\epsilon \in FIRST(Y_1), \ldots, FIRST(Y_{i-1})$,
   add a to $FIRST(X)$
   if $\epsilon \in FIRST(Y_i) \;\; \forall i$, add ϵ to $FIRST(X)$

FOLLOW:

1. Add EOF to $FOLLOW(S)$
2. For each production of the form $A \rightarrow \alpha B \beta$
   (i) add $FIRST(\beta) \backslash \{\epsilon\}$ to $FOLLOW(B)$
   (ii) if $\beta = \epsilon$ or $\epsilon \in FIRST(\beta)$, then add everything in $FOLLOW(A)$ to $FOLLOW(B)$

A Predictive Parser



NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

(Aho,Sethi, Ullman, pp. 186)

36) What are the advantages of using an intermediate language?

Advantages of Using an Intermediate Language

Retargeting - Build a compiler for a new machine by attaching a new code generator to an existing front-end.

2. ***Optimization*** - reuse intermediate code optimizers in compilers for different languages and different machines.

Note: the terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.