UNIT-1 INTRODUCTION TO COMPILERS AND LEXICAL ANALYSIS 9

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

INTRODUCTION TO COMPILERS

Translator:

It is a program that translates one language to another.

Types of Translator:

1.Interpreter
2.Compiler
3.Assembler

1.Interpreter:

It is one of the translators that translate high level language to low level language.

During execution, it checks line by line for errors.
Example: Basic, Lower version of Pascal.

2.Assembler:

It translates assembly level language to machine code.

Example: Microprocessor 8085, 8086.

3.Compiler:

It is a program that translates one language(source code) to another language (target code).

It executes the whole program and then displays the errors.
Example: C, C++, COBOL, higher version of Pascal.

Difference between compiler and interpreter:

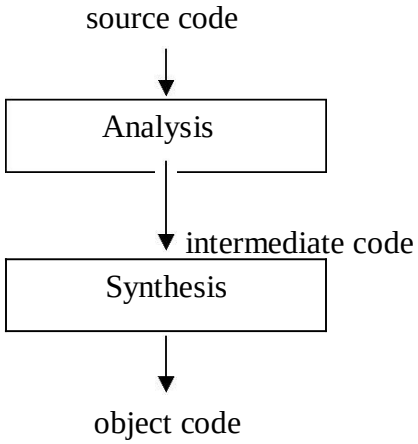
Compiler	Interpreter
It is a translator that translates high level to low level language	It is a translator that translates high level to low level language
It displays the errors after the whole program is executed.	It checks line by line for errors.
Examples: Basic, lower version of Pascal.	Examples: C, C++, Cobol, higher version of Pascal.

PARTS OF COMPILATION

There are 2 parts to compilation:
1. Analysis
2. Synthesis

Analysis part breaks down the source program into constituent pieces and creates anintermediate representation of the source program.

Synthesis part constructs the desired target program from the intermediate representation.



Software tools used in Analysis part:

1) **Structure editor:**

- Takes as input a sequence of commands to build a source program.
- The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- For example , it can supply key words automatically - while …. do and begin….. end.

2) Pretty printers :

- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- For example, comments may appear in a special font.

3) Static checkers :

- A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.

- For example, a static checker may detect that parts of the source program can never be executed.

4) Interpreters :

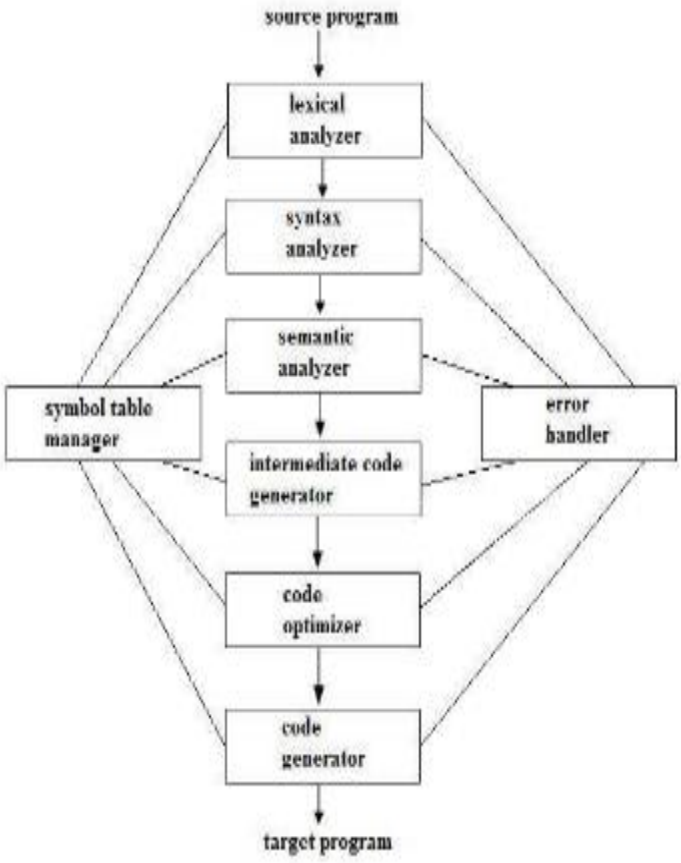
- Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
- An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
- Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

PHASES OF COMPILER

A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:

1) Lexical analysis
2) Syntax analysis
3) Semantic analysis
4) Intermediate code Generation
5) Code optimization
6) Code generation



Sub-Phases:

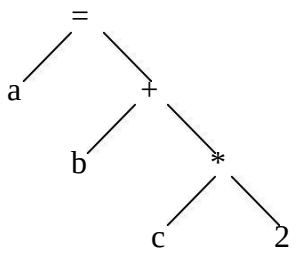
1) Symbol table management
2) Error handling

Lexical analysis:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords,operators, identifiers, special symbols etc.
  Example: a +b =20
  Here, a,b,+,=,20 are all separate tokens.
  Group of characters forming a token is called the **Lexeme**.
- The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

Syntax analysis:

- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.
- Syntax tree:
  It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For a=b+c*2, syntax tree is



Semantic analysis:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

Intermediate code generation:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- The three -address code consists of a sequence of instructions, each of which has atmost three operands.
  Example: t1=t2+t3

Code optimization:

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as

output.

- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

Code generation:

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

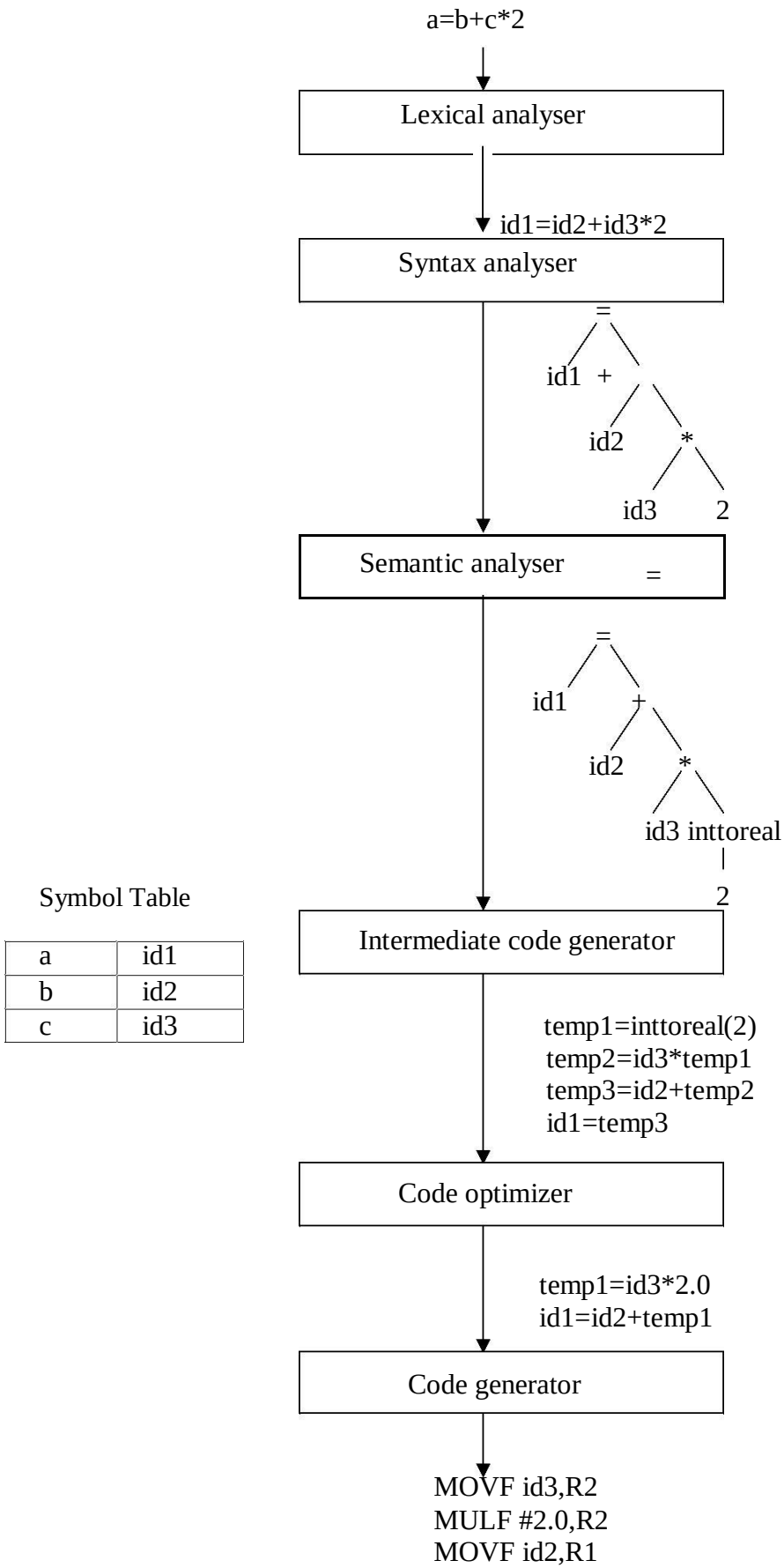
Symbol table management:

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Error handling:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and duringtype conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2. The figure shows the representation of this statement after each phase:

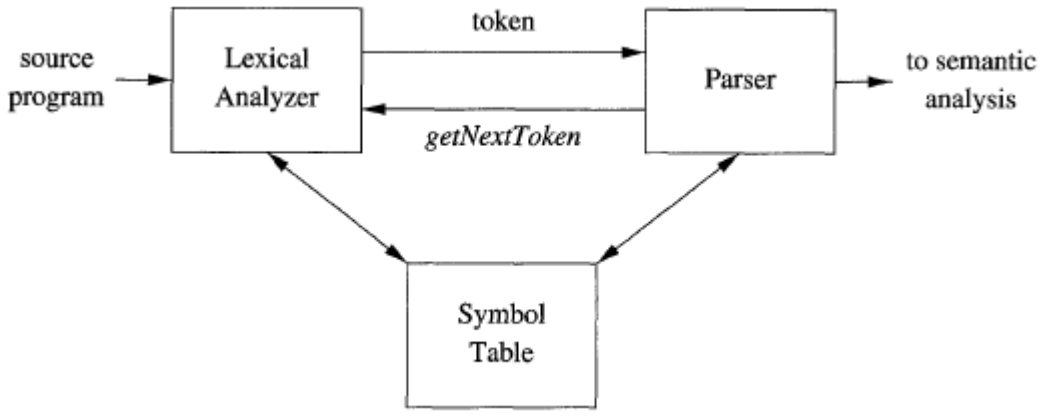

a=b+c*2
Lexical analyser
id1=id2+id3*2
Syntax analyser
=
id1 +
id2 *
id3 2
Semantic analyser
=
=
id1 +
id2 *
id3 inttoreal
2
Symbol Table
a id1
b id2
c id3
Intermediate code generator
temp1=inttoreal(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
Code optimizer
temp1=id3*2.0
id1=id2+temp1
Code generator
MOVF id3,R2
MULF #2.0,R2
MOVF id2,R1

LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

THE ROLE OF THE LEXICAL ANALYZER

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.

One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also
be performed by the lexical analyzer.

Issues in lexical analysis:

1. **Simplicity of design** : It is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments
and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. **Compiler efficiency is improved**: A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input
characters can speed up the compiler significantly.
3. **Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

Tokens, patterns, and lexemes:

When discussing lexical analysis, we use three related but distinct terms:

Token:

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

Pattern:

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token,the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens,
the pattern is a more complex structure that is *matched* by many strings.

Lexeme:

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

1.Tokens are treated as terminal symbols in the grammar for the source language using boldface names to represent tokens.
2.The lexemes matched by the pattern for the tokens represent the strings of characters in the source program that can be treated together as a lexical unit
3.In most of the programming languages keywords, operators identifiers , constants , literals and punctuation symbols are treated as tokens.
4. A pattern is a rule describing the set of lexemes that can represent a particular token in the source program.
5. In many languages certain strings are reserved i.e their meanings are predefined and cannot be changes by the users
6. If the keywords are not reserved then the lexical analyzer must distinguish between a keyword and a user-defined identifier

Attributes for tokens:
When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse. We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with
the token a great deal of information. Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example : The token names and associated attribute values for the Fortran statement
are written below as a sequence of pairs.
<id, pointer to symbol-table entry for E>
< assign-op >
<id, pointer to symbol-table entry for M>
<mult -op>
<id, pointer to symbol-table entry for C>
<exp-op>
<number , integer value 2 >
Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

Errors in lexical analysis: It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string f i is encountered for the first time in a C program in the context: a lexical analyzer cannot tell whether f i is a misspelling

of the keyword if or an undeclared function identifier. Since f i is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - probably the parser in this case - handle an error due to transposition of the letters. However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

Panic mode :

The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

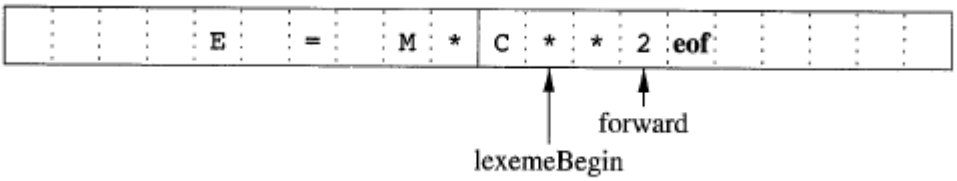
3. Replace a character by another character.

4. Transpose two adjacent characters.

INPUT BUFFERING:

During lexical analyzing , to identify a lexeme , it is important to look ahead atleast one additional character. Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character

An important scheme involves two buffers that are alternatively reloaded



Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

I. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig. 3.3, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

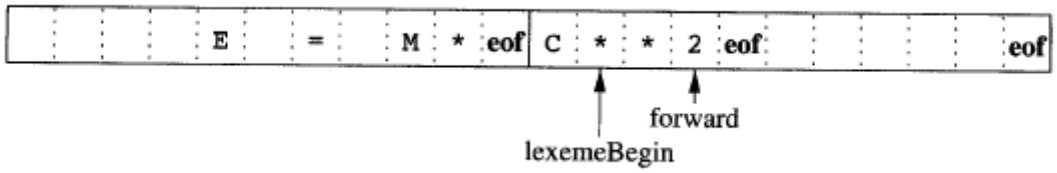
Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else forward := forward + 1;
```

Fig. 3.4. Code to advance forward pointer.

Sentinels

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 3.5 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.




```
forward := forward + 1;
if forward↑ = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward + 1
        end
        else if forward at end of second half then begin
                reload first half;
                move forward to beginning of first half
        end
        else /* eof within a buffer signifying end of input */
                terminate lexical analysis
end
```

Fig. 3.6. Lookahead code with sentinels.

SPECIFICATION OF TOKENS :

Regular languages are an important notation for specifying lexeme patterns

Strings and Languages:

- An alphabet is any finite set of symbols ex: Letters, digits and punctuation
- The set {01) is the binary alphabet
- A string over an alphabet is a finite sequence of symbols drawn from the alphabet
- The length of the string s,represented as |s| , is the number of occurrences of symbols in s .
- The empty string denoted as € is the string of length 0
- A language is any countable set of strings over some fixed alphabet ex: abstract languages
- If x and y are strings ten the concatenation of x and y denoted by xy is the string formed by appending y to x for ex if x=cse and y=department , then xy=csedepartment .

Operation on Languages:

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (Kleene) closure of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times. Note that Lo, the "concatenation of L zero times," is defined to be {E), and inductively, L~ is Li-'L. Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term Lo. That is, E will not be in L+ unless it is in L itself.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Example ': Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z) and let D be the set of digits {0,1,.. .9). We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of the Fig:

1. L U D is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set df 520 strings of length two, each consisting of one letter followed by one digit.
3. L4 is the set of all 4-letter strings.
4. L* is the set of ail strings of letters, including e, the empty string.

5. L(L U D)* is the set of all strings of letters and digits beginning with a letter.
6. D+ is the set of all strings of one or more digits.

Regular expressions:

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if letter- is established to stand for any letter or the underscore, and digit- is established to stand for any digit, then we could describe the language of C
identifiers by:
letter- (letter- I digit)*
The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of letter- with the remainder of the expression signifies concatenation. The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet C and the languages that those expressions denote.

BASIS: There are two rules that form the basis:
1. E is a regular expression, and L (E) is {E) , that is, the language whose sole member is the empty string.
2. If a is a symbol in C, then a is a regular expression, and L(a) = {a), that is, the language with one string, of length one, with a in its one position.

Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.'

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.
1. (r)1 (9) is a regular expression denoting the language L(r) U L(s).
2. (r) (s) is a regular expression denoting the language L(r) L(s) .
3. (r) * is a regular expression denoting (L (r)) * .
4. (r) is a regular expression denoting L(r). This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote. As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:
a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression (a) I ((b) * (c)) by a/ b*c. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Example : Let C = {a, b}.
1. The regular expression a1 b denotes the language {a, b}.

2. (a1 b) (alb) denotes {aa, ab, ba, bb), the language of all strings of length two over the alphabet C. Another regular expression for the same language is aalablbal bb.
3. a* denotes the language consisting of all strings of zero or more a's, that is, {E, a, aa, aaa, . . . }.
4. (alb)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {e, a, b, aa, ab, ba, bb, aaa, . . .}. Another regular expression for the same language is (a*b*)*.
5. ala*b denotes the language {a, b, ab, aab,aaab,. . .), that is, the string a and all strings consisting of zero or more a's and ending in b. A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (alb) = (bla). There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure below shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If C is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

where:
1. Each di is a new symbol, not in C and not the same as any other of the d's
2. Each ri is a regular expression over the alphabet C U {dl, d2,. . . , di-l).

By restricting ri to C and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over C alone, for each ri. We do so by first replacing uses of dl in r2 (which cannot use any of the d's except for dl), then replacing uses of dl and d2 in r3 by rl and (the substituted) 7-2, and so on. Finally, in rn we replace each di, for i = 1,2,. . . ,n - 1, by the substituted version of ri, each of which has only symbols of C.

Example : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

letter- + A (B I . - . [Z 1 a 1 b l . - - l z 1 -

digit -+ 01 1 1 - - . 1 9
id + letter- (letter- I digit)*

Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns.

Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regularexpression variants in use today.

1. One or more instances. The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then (r)+d enotes the language (~ (r)) 'T. he operator
has the same precedence and associativity as the operator *. Two useful algebraic laws, r* = r+Jc and rf = rr* = r*r relate the Kleene closure and positive closure.

2. Zero or one instance. The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to rlc, or put another way, L(r?) = L(r) U (€1. The ? operator has the same precedence and associativity as
* and +.

3. Character classes. A regular expression allazl. .. lan, where the ai's are each symbols of the alphabet, can be replaced by the shorthand [ala2 . . . a,]. More importantly, when a1 , a2, . . . , a, form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by al-a,, that is, just the first and last separated by a hyphen. Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a | b |. . . |z.

RECOGNITION OF TOKENS:

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

stmt -> **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| other

expr -> *term* **relop** *term*
| *term*

term -> **id**
| **number**

This syntax is similar to that of the language Pascal, in that then appears explicitly after conditions.

For relop, we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes. The terminals of the grammar, which are if, then, else, relop, id, and number, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

For this language, the lexical analyzer will recognize the keywords i f , then, and else, as well as lexemes that match the patterns for relop, id, and number. To simplify matters, we make the common assumption that keywords are also reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers. In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" ws defined by:

ws -+ (blank I tab (newline)+

Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Transition diagram:

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams.

State:

Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges:

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled
by a (and perhaps by other symbols, as well). If we find such an edge, we advance the forward pointer arid enter the state of the transition diagram to which that edge leads.

We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer.

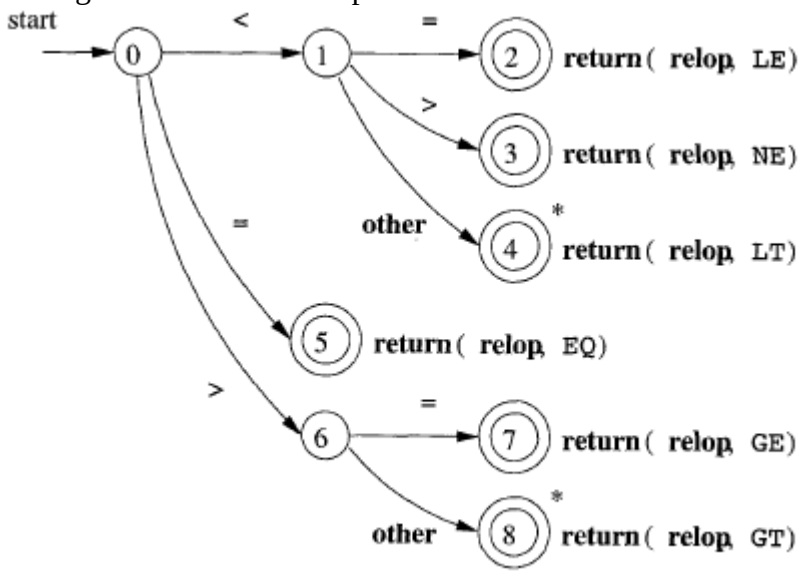
Some important conventions about transition diagrams are:

1.Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the LexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the start state, or initial state; it is indicated by an edge, labeled "start ," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

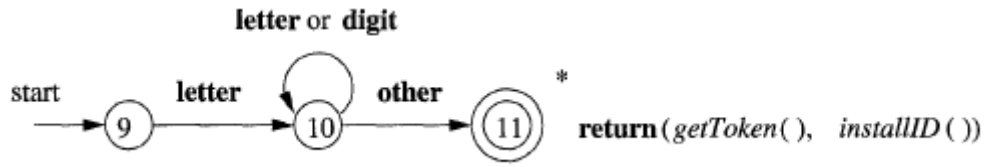
Example: Transition diagrams for relational operations.



Recognizing keywords and identifiers :

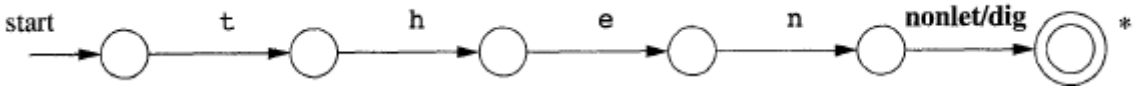
Recognizing keywords and identifiers presents a problem. Usually, keywords like if or then are reserved (as they are in our running example), so they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords i f , then, and else of our running example.

letter or digit:



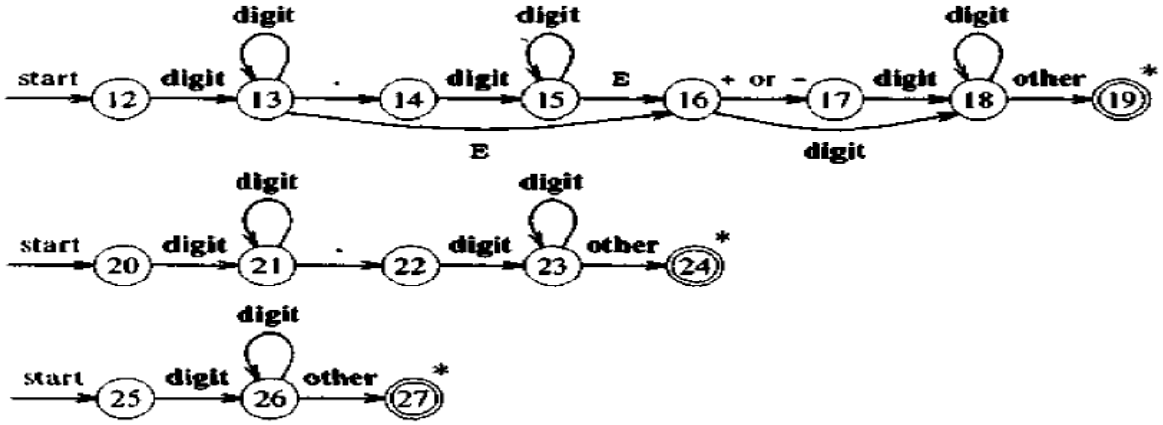
A transition diagram for id's and keywords

There are two ways that we can handle reserved words that look like identifiers:
1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function getToken examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.
2. Create separate transition diagrams for each keyword; an example for the keyword then is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like thenext value that has then as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.



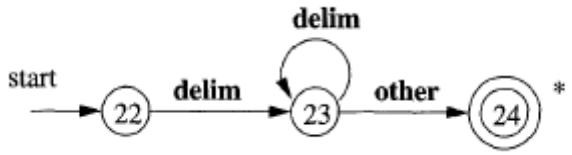
Hypothetical transition diagram for the keyword then

The transition diagram for a token number is given below



- If a dot is seen we have an fraction number. Three diagrams in which first one is the recognition of number which has fractional and exponent part or only exponent part.
- Second diagram is the recognition of number which has only fractional part.
- Third diagram is the recognition of number which has only decimal value.

A transition diagram for whitespace is given below



In the diagram we look for one or more whitespace characters represented by delim in the diagram –typically these characters would be blank,tab,newline

Implementing a Transition Diagram

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagrams. We adopt a systematic approach that works for all transition diagrams and constructs programs whose size is proportional to the number of states and edges in the diagrams.

Each state gets a segment of code. If there are edges leaving a state, then its code reads a character and selects an edge to follow, if possible. A function `nextchar()` is used to read the next character from the input buffer, advance the forward pointer at each call, and return the character read.[3] If there is an edge labeled by the character read, or labeled by a character class containing the character read, then control is transferred to the code for the state pointed to by that edge. If there is no such edge, and the current state is not one that indicates a token has been found, then a routine `fail()` is invoked to retract the forward pointer to the position of the beginning pointer and to initiate a search for a token specified by the next transition diagram. If there are no other transition diagrams to try, `fail()` calls an error-recovery routine.

To return tokens we use a global variable `lexical_value`, which is assigned the pointers returned by functions `install_id()` and `install_num()` when an identifier or number, respectively, is found. The token class is returned by the main procedure of the lexical analyzer, called `nexttoken()`.

We use a case statement to find the start state of the next transition diagram. In the C implementation in Fig. 3.15, two variables `state` and `start` keep track of the present state and the starting state of the current transition diagram. The state numbers in the code are for the transition diagrams of Figures 3.12 – 3.14.

```
int state = 0, start = 0;
int lexical_value;
     /* to "return" second component of token */

int fail()
{
     forward = token_beginning;
     switch (start) {
         case 0:    start = 9; break;
         case 9:    start = 12; break;
         case 12:   start = 20; break;
         case 20:   start = 25; break;
         case 25:   recover(); break;
         default:    /* compiler error */
     }
     return start;
}
```

Fig. 3.15. C code to find next start state.

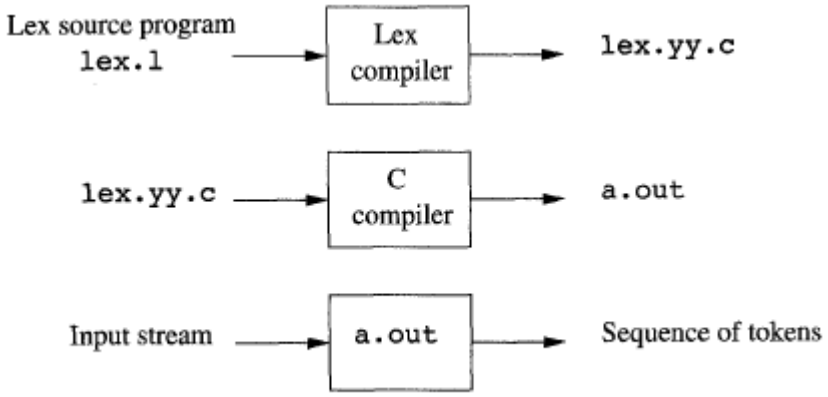
```
token nexttoken()
{    while(1) {
         switch (state) {
         case 0:    c = nextchar();
             /* c is lookahead character */
             if (c==blank || c==tab || c==newline) {
                 state = 0;
                 lexeme_beginning++;
                     /* advance beginning of lexeme */
             }
             else if (c == '<') state = 1;
             else if (c == '=') state = 5;
             else if (c == '>') state = 6;
             else state = fail();
             break;

             .../* cases 1-8 here */
```

THE LEXICAL- ANALYZER GENERATOR - LEX

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex . yy . c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

Use of Lex:

An input file lex1 is written in the lex language and describes the lexical analyzer to be generated . The Lex compiler transforms lex1 to a c program in a file that is always named lex.yy.c



Structure of a lex program:

A Lex program has the following form:
declarations
%%
translation rules
%%
auxiliary functions

The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions

The translation rules each have the form

Pattern { Action)

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C.

The third section holds whatever additional functions are used in the actions.

Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by Lex behaves as follows :

1. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns Pi.

2.It then executes the associated action Ai. Typically, Ai will return to the parser, but if it does not (e.g., because Pi describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
3.The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

Lex program for token :

```
%{
/* definitions of manifest constants
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim [\t\nl
ws (delim)+
letter [A-Za-z]
digit [o-9]
id {letter} {(letter) | {digit})*
number {digit)+ (\. {digit}+)? (E [+-] ?{digit}+)?

%%
{ws} (/* no action and no return */)
if    {return(IF) ;}
then  {return(THEN) ;}
else  {return(ELSE) ;}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum() ;return(NUMBER) ;}
"<"   {yylval = LT; return(REL0P);}
"<="  {yylval = LE; return(REL0P);}
"="   {yylval = EQ ;return(REL0P);}
"<>"  {yylval = NE; return(REL0P);}
">"   {yylval = GT; return(REL0P);}
">="  {yylval = GE; return(REL0P);}
%%
int installID0 {/* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng, into the
                  symbol table and return a pointer
                  thereto */
}
int installNum() {/* similar to installID, but puts numerical
                  constants into a separate table */
}
```

In the declarations section we see a pair of special brackets, %(and %). Anything within these brackets is copied directly to the file lex . yy . c, and is not treated as a regular definition.The manifest constants are placed inside it
Also the languages occur as a sequence of regular definitions .
Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, delim is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively

In the auxiliary-function section, we see two such functions, installID( )and installNum() . Like the portion of the declaration section that appears between everything in the auxiliary section is copied directly to file lex. yy . c, but may be used in the actions.

First, ws, an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern

i f . Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for id), then the lexical analyzer consumes these two letters from the input and returns the token name IF, that is, the integer for which the manifest constant IF stands. Keywords then and else are treated similarly. The fifth token has the pattern defined by id. Note that, although keywords like i f match this pattern as well as an earlier pattern, Lex chooses whichever pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when id is matched is given as follows:

I. Function installID( ) is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler. Note that installID () has available to it two variables that are set automatically by the lexical analyzer that Lex generates:

(a) yytext is a pointer to the beginning of the lexeme

(b) yyleng is the length of the lexeme found.

3. The token name I D is returned to the parser.

FINITE AUTOMATA:

We shall now discover how Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as finite automata. These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
2. Finite automata come in two flavors:

(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and E, the empty string, is a possible label.

(b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

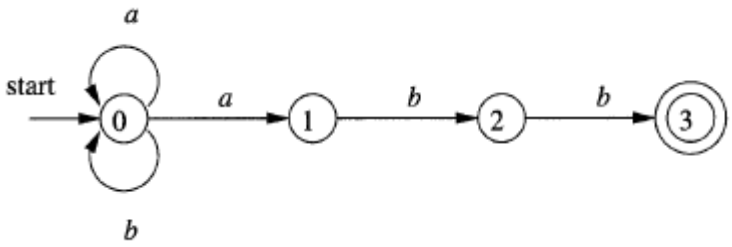
Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the regular languages, that regular expressions can describe

NDFA:

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols C, the input alphabet. We assume that E, which stands for the empty string, is never a member of C.
3. A transition function that gives, for each state, and for each symbol a set of next states.
4. A state so from S that is distinguished as the start state (or initial state).
5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a

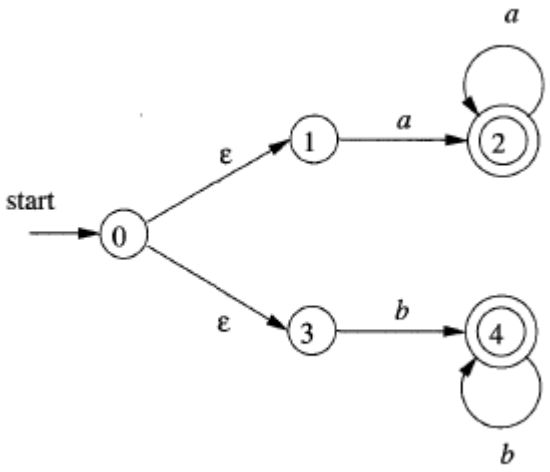


DFA:

A deterministic finite automaton (DFA) is a special case of an NFA where:

1. There are no moves on input
2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets. While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.



MINIMIZATION OF DFA:

Two method of solving

1) **by using table filling algorithm**

   Example:

Minimize the given regular expression (a|b)*abb.

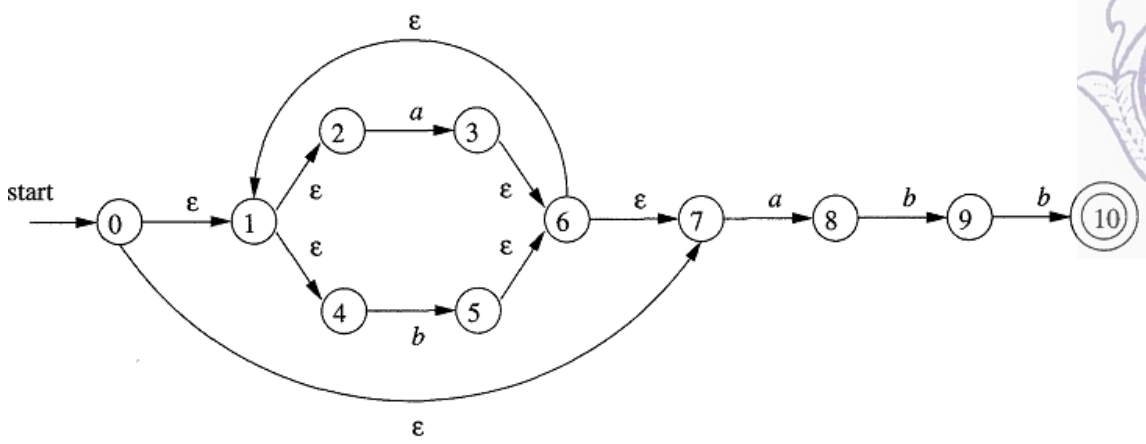


Figure 3.34: NFA N for (a|b)*abb

ε – closure (0) = {0, 1, 2, 4, 7 } = A

Dtran[A, a] = ε – closure

(move(A, a))

= ε – closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8 }

= B

Dtran[A, b] = ε – closure (move(A, b))

= ε – closure (5)

= {1, 2, 4, 5, 6, 7 }

= C

Dtran[B, a] = ε – closure (move(B, a))

= ε – closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8 }

= B

Dtran[B, b] = ε – closure (move(B, b))

= ε – closure (5, 9)

= {1, 2, 4, 5, 6, 7, 9 }

= D

Dtran[C, a] = ε – closure (move(C, a))

= ε – closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8 }

= B

Dtran[C, b] = ε – closure (move(C, b))

= ε – closure (5)

= {1, 2, 4, 5, 6, 7 }

= C

Dtran[D, a] = ε – closure (move(D, a))

= ε – closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8 }

= B

Dtran[D, b] = ε – closure (move(D, b))

= ε – closure (5, 10)

= {1, 2, 4, 6, 7, 10 }

= E

Dtran[E, a] = ε – closure (move(E, a))

= ε – closure (3, 8)

= {1, 2, 3, 4, 6, 7, 8 }

= B

Dtran[E, b] = ε – closure (move(E, b))

= ε – closure (5)

= {1, 2, 4, 5, 6, 7 }

= C

NFA State	DFA State	a	b
{0, 1, 2, 4, 7 }	A	B	C
{1, 2, 3, 4, 6, 7, 8 }	B	B	D
{1, 2, 4, 5, 6, 7 }	C	B	C
{1, 2, 4, 5, 6, 7, 9 }	D	B	E
{1, 2, 4, 6, 7, 10 }	E	B	C

Fig. Transition table Dtran for DFA D

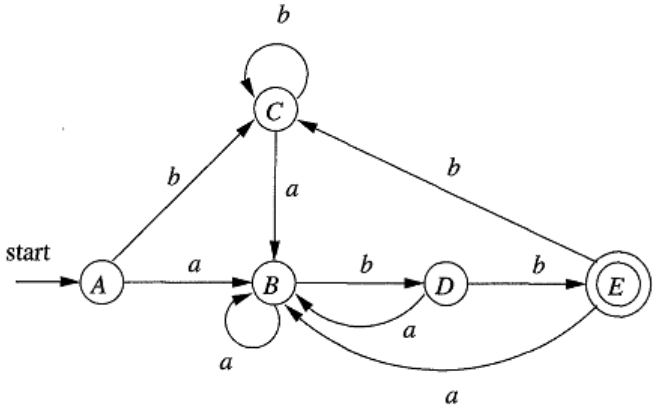


Figure 3.36: Result of applying the subset construction to

Finally optimize the above one by using table filling algorithm. Refer Class notes.

2) by using direct method

- By concatenating a unique right endmarker # to a regular expression r, we give the accepting state for r a transition on #, making it an important state of the NFA for (r)#.
- By using the augmented regular expression (r)#, any state with a transition on #

must be an accepting state.

- to present the regular expression by its syntax tree, where the leaves correspond to operands and the interior nodes correspond to operators.

An interior node is called

a cat-node is labeled by the concatenation operator (dot) or-node is labeled by union operator |

star-node is labeled by star operator *

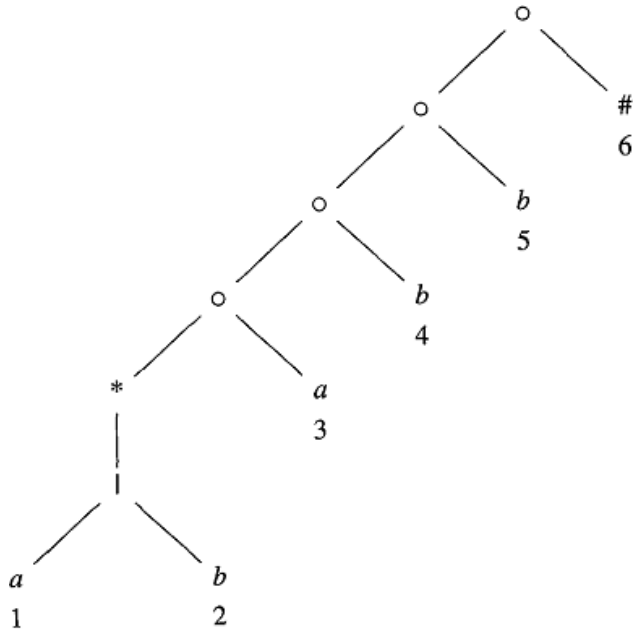


Figure 3.56: Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

Leaves in a syntax tree are labeled by ε or by an alphabet symbol. To each leaf not labeled ε, we attach a unique integer. We refer to this integer as the position of the leaf and also as a position of its symbol.

Functions Computed From the Syntax Tree:

To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: nullable, firstpos, lastpos, and followpas.

Each definition refers to the syntax tree for a particular augmented regular expression (r) #.

1. *nullable(n)* is true for a syntax-tree node n if and only if the subexpression

represented by n has ε in its language.

2. *firstpos(n)* is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.
3. *lastpos(n)* is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.
4. *followpos(p),* for a position p, is the set of positions q in the entire syntax tree such that there is some string x = $a_1a_2\ldots a_n$, in L ((r)#) such that for some i, there is a way to explain the membership of x in L((r)#) by matching a_i to position p of the syntax tree and a_{i+1} to position q.

Example:

The expression (a|b)*a. We claim nullable(n) is false, since this node generates all strings of a's and b's ending in an a; it does not generate ε.

firstpos(n) = {1,2,3)

lastpos(n) = (3)

followpos(1) = {1,2,3)

Computing nullable, firstpos, and lastpos:

The basis & induction rules for nullable(n), firstpos(n) lastpos(n) & followpos(n) is as follows.

Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
Leaf ε	true	∅	∅
Leaf *i*	false	{*i*}	{*i*}
	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
	true	*firstpos*(c_1)	*lastpos*(c_1)

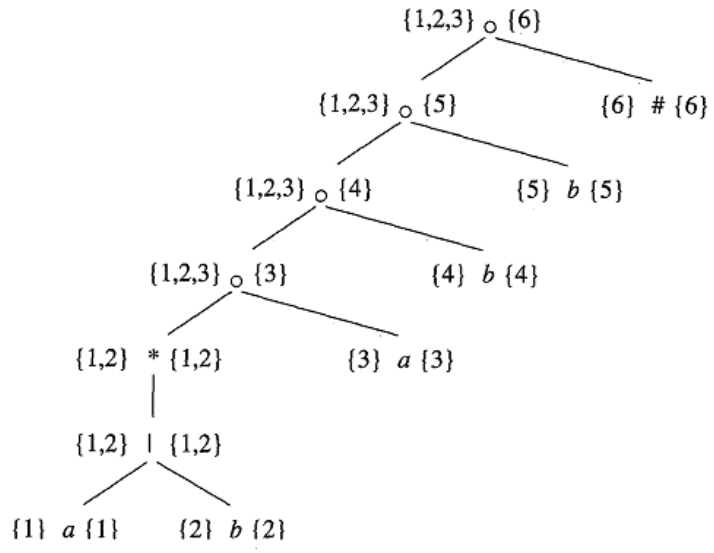


Figure 3.59: *firstpos* and *lastpos* for nodes in the syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

Computing *followpos*:

There are only two ways that a position of a regular expression can be made to follow another.

1. If n is a cat-node with left child C1 and right child C2, then for every position i in *lastpos(c1),* all positions in *firstpos(c2)* are in *followpos(i).*
2. If n is a star-node, and i is a position in *lastpos(n),* then all positions in *firstpos(n)* are in *followpos(i)* .

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Figure 3.60: The function *followpos*

The *followpos* is almost an NFA without ε-transitions for the underlying regular expression, and would become one if we:

1. Make all positions in firstpos of the root be initial states,
2. Label each arc from i to j by the symbol at position i, and
3. Make the position associated with endmarker # be the only accepting state.

Compilation and interpretation:

Compilation and interpretation are two different approaches to executing computer programs. They are fundamental concepts in computer science and programming. Let's explore each of them:

Compilation:

1. **Process:** In compilation, the source code written by the programmer is translated into machine code or an intermediate code by a compiler before it is executed. The compiler reads the entire source code and generates an executable file or binary code, which can be run multiple times without the need for recompilation (unless the source code is modified).

2. **Speed:** Compiled programs generally execute faster than interpreted programs because the translation from source code to machine code happens ahead of time. This means that the compilation process optimizes the code for the target platform.

3. **Examples:** C, C++, and Rust are examples of programming languages that are typically compiled. These languages produce standalone executable files.

4. **Advantages:** Compilation can catch syntax errors and some semantic errors in the code before the program runs. It also allows for performance optimizations, making it suitable for applications where speed is critical.

5. **Disadvantages:** Compilation can be time-consuming, especially for large programs, as the entire code needs to be translated before execution. It may also be less flexible than interpretation because changes to the source code require recompilation.

Interpretation:

1. **Process:** In interpretation, the source code is executed directly by an interpreter line by line, without generating a separate executable file. The interpreter reads the code, processes it, and executes it in real-time.

2. **Speed:** Interpreted programs are generally slower than compiled programs because there is no optimization phase where the entire code is translated into machine code beforehand.

3. **Examples:** Python, Ruby, and JavaScript are examples of languages that are typically interpreted. These languages often use an interpreter to run the code directly from the source.

4. **Advantages:** Interpretation provides more flexibility because changes to the source code take effect immediately without the need for recompilation. It is also often easier for debugging since errors are detected and reported as they occur.

5. **Disadvantages:** Interpreted programs may run slower than compiled ones due to the lack of pre-optimization. Additionally, syntax and semantic errors might not be caught until runtime, potentially leading to unexpected behavior during execution.

In practice, some programming languages and environments offer a combination of both compilation and interpretation. For example, Java uses a combination of compilation to bytecode (which is then interpreted by the Java Virtual Machine) to achieve platform independence. Other languages, like C# in the .NET framework, use a just-in-time (JIT) compiler to translate bytecode into machine code at runtime for performance reasons.

The choice between compilation and interpretation depends on factors like the specific programming language, the project's requirements, performance considerations, and development preferences.

Phases Of Compiler:

A compiler is a complex piece of software that translates source code written in a high-level programming language into machine code or some other lower-level representation that can be executed by a computer. The compilation process typically consists of several distinct phases, each responsible for specific tasks. Here are the standard phases of a compiler:

1. **Lexical Analysis (Scanning):**
   - The first phase, often called lexical analysis or scanning, reads the source code character by character.
   - It breaks the input into smaller units called tokens, which are the basic building blocks of the language (e.g., keywords, identifiers, literals, and operators).
   - Lexical analysis also discards comments and whitespace.

2. **Syntax Analysis (Parsing):**
   - Syntax analysis, also known as parsing, takes the stream of tokens generated by the lexical analysis and organizes them into a hierarchical structure called a syntax tree or abstract syntax tree (AST).
   - This phase checks if the source code adheres to the grammar rules of the programming language.
   - It identifies the relationships between different parts of the code, such as function calls, variable declarations, and control structures.

3. **Semantic Analysis:**
   - Semantic analysis checks the meaning of the code beyond its syntax.
   - It enforces type checking to ensure that variables and expressions have compatible types, and it verifies other language-specific rules.
   - This phase also resolves references between variables and functions (e.g., scope resolution).

4. **Intermediate Code Generation:**
   - Some compilers generate an intermediate representation of the code after syntax and semantic analysis. This intermediate code is often platform-independent and easier to optimize.
   - The intermediate code generation phase simplifies the compilation process and enables various optimizations.

5. **Optimization:**
   - The optimization phase is responsible for improving the efficiency of the code generated by the compiler.
   - It applies various transformations to the intermediate code or directly to the target code to make it run faster and use fewer resources.
   - Optimization techniques can vary widely, from basic optimizations like constant folding to more advanced techniques like loop unrolling and inlining.

6. **Code Generation:**
   - The code generation phase translates the optimized or intermediate code into machine code or another lower-level representation specific to the target platform.
   - It handles memory management, registers allocation, and instruction selection, tailoring the code for the target architecture.
   - Code generation ensures that the program can be executed on the target machine.

7. **Symbol Table Management:**
   - Throughout the compilation process, a symbol table is maintained. It stores information about variables, functions, and other program entities, helping with scope resolution, type checking, and code generation.

8. **Error Handling:**
   - The compiler must handle errors gracefully. It reports syntax errors, semantic errors, and other issues to the programmer, providing information about the location and nature of the errors.

9. **Output:**
   - Finally, the compiler produces the output, which may be an executable file, an object file, or some other representation suitable for the target platform.

These phases may not always be strictly sequential, as some compilers use techniques like incremental compilation to speed up the process or apply optimizations at various stages. Nonetheless, understanding these phases helps clarify the complex task a compiler performs in translating high-level code into executable machine code.

Language for Specifying Lexical Analyzers:

Lexical analyzers (also known as lexers or scanners) are responsible for breaking down the input source code into a sequence of tokens for further processing by a compiler or interpreter. To specify the rules for lexers, you often use specialized languages or tools designed for this purpose. Here are some common languages and tools used to specify lexical analyzers:

1. **Regular Expressions:**
   - Regular expressions are a powerful and widely used method for specifying lexical rules. They describe patterns of characters, making it easy to recognize tokens like keywords, identifiers, numbers, and symbols.

- Popular programming languages like Python, Java, and C++ support regular expressions for defining lexical rules.

2. **Lex (Lexical Analyzer Generator):**
   - Lex is a widely used lexer generator that allows you to specify lexical rules using regular expressions. It generates the corresponding C code for the lexer.
   - Flex is a popular variant of Lex and is often used in Unix-based environments.

3. **ANTLR (ANother Tool for Language Recognition):**
   - ANTLR is a powerful parser generator that can also be used for defining lexers. It uses a specialized grammar notation to specify lexer rules.
   - ANTLR can generate lexers and parsers in various programming languages, including Java, C#, and Python.

4. **JFlex:**
   - JFlex is a lexer generator for Java that enables you to specify lexical rules using regular expressions and Java code embedded in special sections.
   - It generates efficient Java code for the lexer.

5. **Ragel:**
   - Ragel is a state machine compiler that allows you to define lexers (and parsers) using a domain-specific language for state machines.
   - It can generate code in various programming languages, including C, C++, and Ruby.

6. **Lexical Specification in Programming Languages:**
   - Some programming languages have built-in support for specifying lexical rules. For example, in the Haskell programming language, you can define lexers using parser combinators or monadic parsers.

7. **Custom Lexer Specification:**
   - In some cases, you might write custom lexer specifications by hand, especially for simple languages or when using programming languages that lack built-in lexer generators.
   - This involves writing code that reads the input characters, applies lexical rules, and generates tokens.

When choosing a method or tool for specifying lexical analyzers, consider factors like the complexity of the language you're working with, the target programming language for your compiler or interpreter, and your familiarity with the tools available. Lexer generators like Lex, Flex, and ANTLR are particularly useful for creating robust and efficient lexers for complex languages, while regular expressions are suitable for simpler tasks or quick prototyping.

UNIT II SYNTAX ANALYSIS 12

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar – Top Down Parsing - General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table - Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC.

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

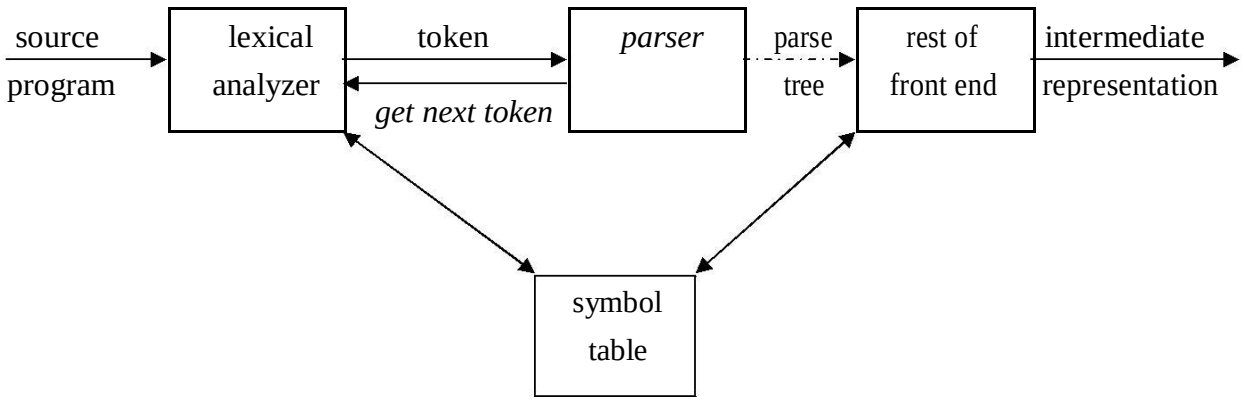
Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.

4. It performs error recovery.

Issues :

Parser cannot detect errors such as:
1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling :

Programs can contain errors at many different levels. For example :
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse

tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar: The following grammar defines **simple** arithmetic expressions:

expr → *expr op expr*
expr → (*expr*)
expr → - *expr*
expr → **id**
op → +
op → -
op → *
op → /
op → ↑

In this grammar,

- **id** + - * / ↑( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

Derivations:

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

E → E+E |E*E |(E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.

- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E |E*E |(E) | - E |id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal

or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

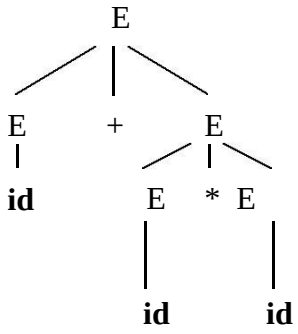
Ambiguity:

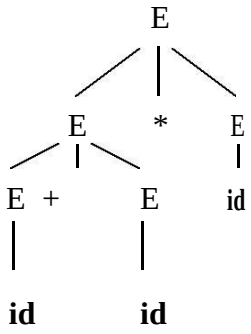
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id





WRITING A GRAMMAR

There are four categories in writing a grammar:

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

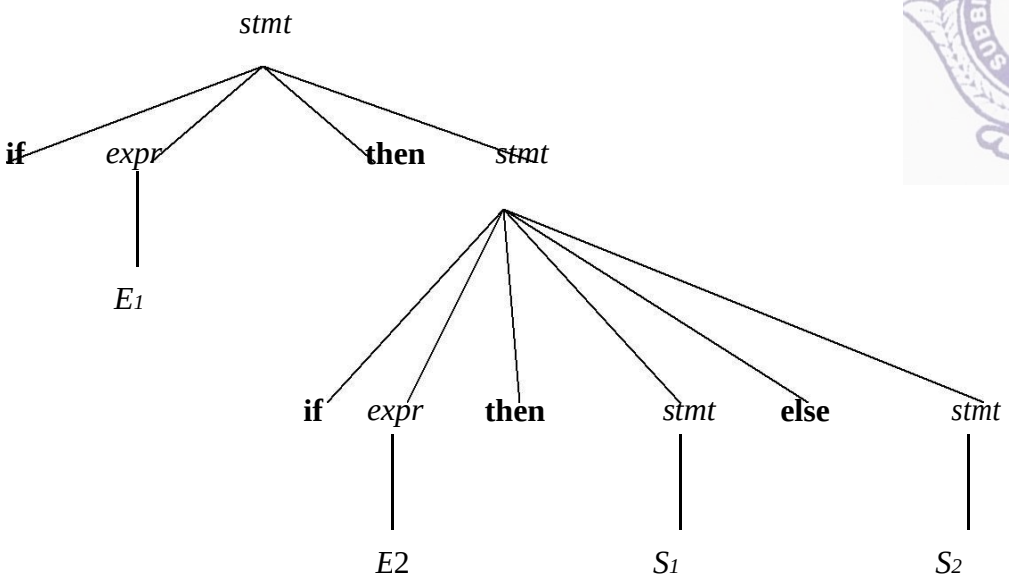
- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

Eliminating ambiguity:

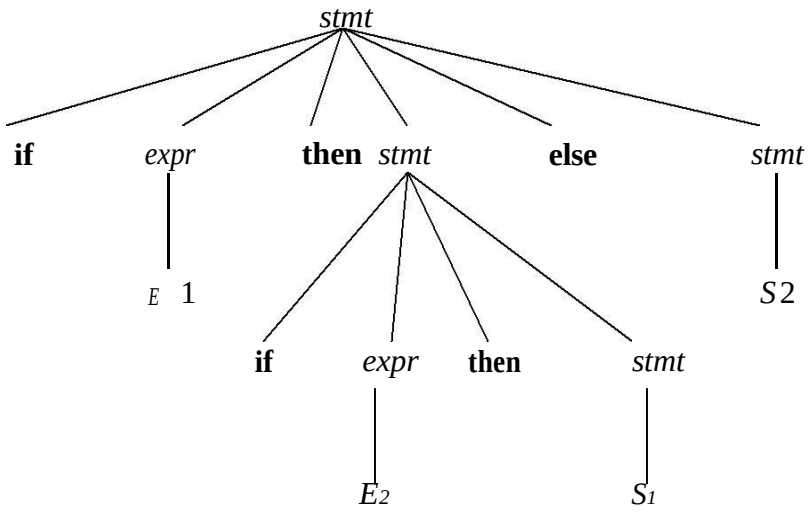
Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if** E_1 **then if** E_2 **then** S_1 **else** S_2 has the following two parse trees for leftmost derivation:



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα |β it can be replaced with a sequence of two productions

A → βA'

A'→ αA' | ε without

changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T |T

T → T*F |F

F→ (E) |id

First eliminate the left recursion for E

as E → TE'

E' → +TE' |ε

Then eliminate for T

as T → FT'

T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion

is E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) |id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. **for** *i* := 1 **to** *n* **do begin**
   for *j* := 1 **to** *i*-1 **do begin**
   replace each production of the form $A_i \rightarrow A_j \gamma$ by
   the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions;
   end
   eliminate the immediate left recursion among the A_i-productions
   end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar , G : S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS' | a
S'→ eS |ε
E → b

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.
  Example : LL Parsers.
- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
  Example : LR Parsers.

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar G : S → cAd

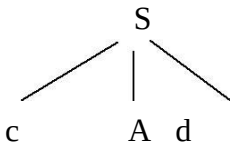
A → ab |a

and the input string w=cad.

The parse tree can be constructed using the followingtop-down approach :

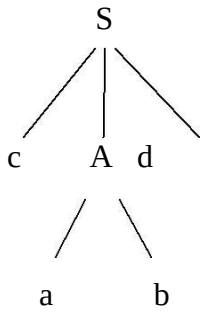
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



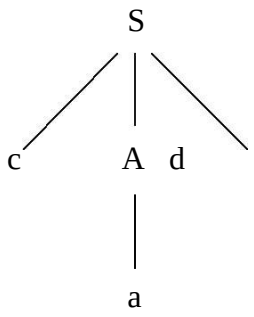
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

PREDICTIVE PARSER:

It is implemented by using non backtracking concept. It is categorized into two types,

1. recursive Predictive parser
2. non recursive Predictive parser

RECURSIVE PREDICTIVE PARSER

It is implemented by using recursion mechanism. A left-recursive grammar can cause a recursive Predictive parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After eliminating the left-recursion the grammar

becomes, $E \rightarrow TE'$

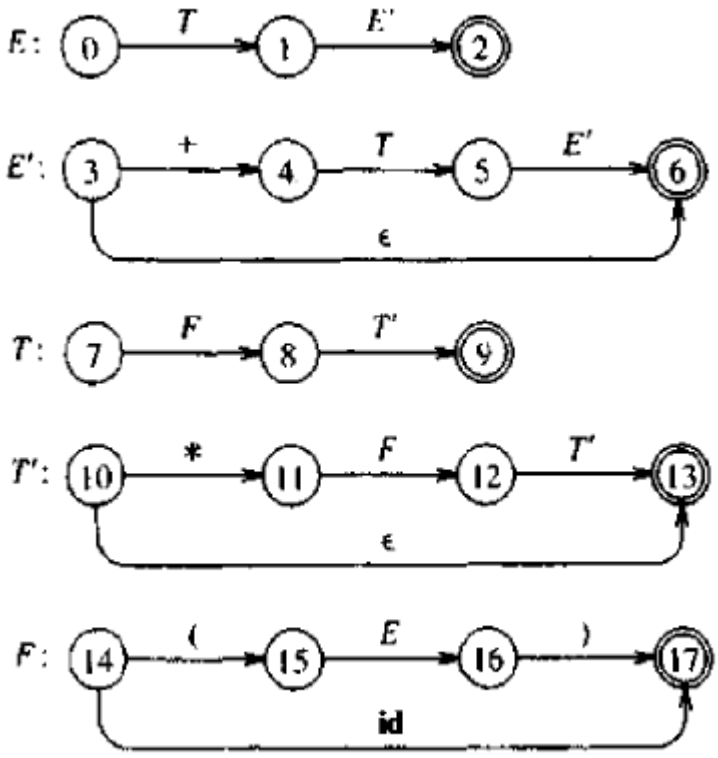
$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

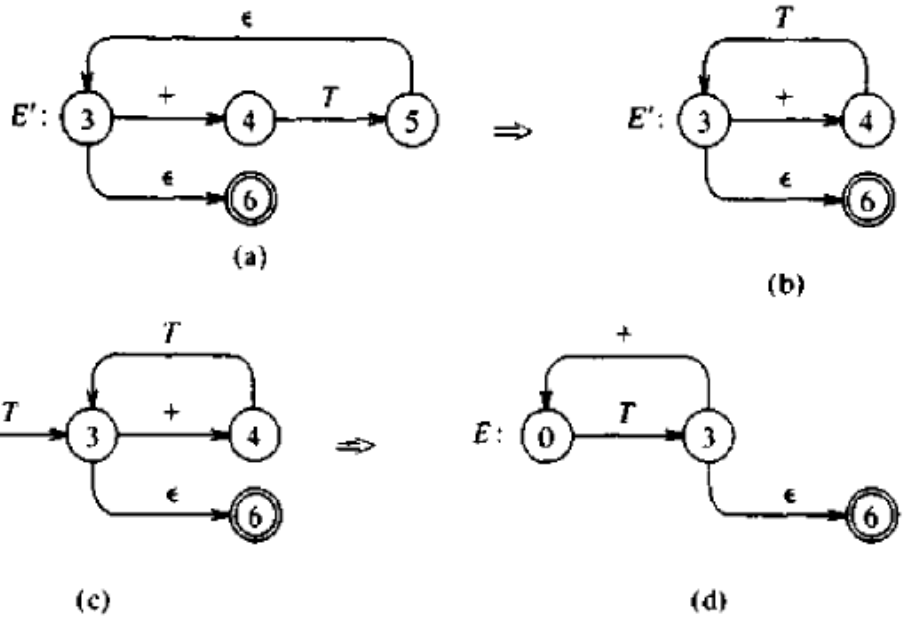
$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

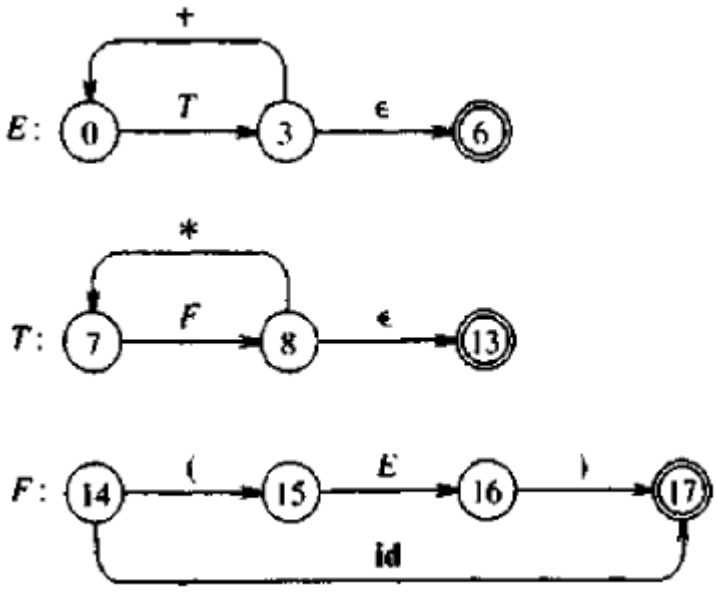
Now we can draw transition diagram for each of the non terminal,



From the above diagram we can reduce the number of states in E' diagram and substitute it in Nonterminal E diagram as follows,



Similarly reducing T' Diagram, the resultant transition diagram are,



write the procedure based on above grammar as follows:

Recursive procedure:

```
Procedure E()
begin
        T( );
        EPRIME( );
end


Procedure EPRIME( )

begin
        If input_symbol='+' then ADVANCE( );
        T( ); EPRIME( );

End
Procedure T( )
 begin
        F( ); TPRIME( );
end
Procedure TPRIME( )
begin
        If input _symbol='*' then ADVANCE(
        );
        F( ); TPRIME(  );
```

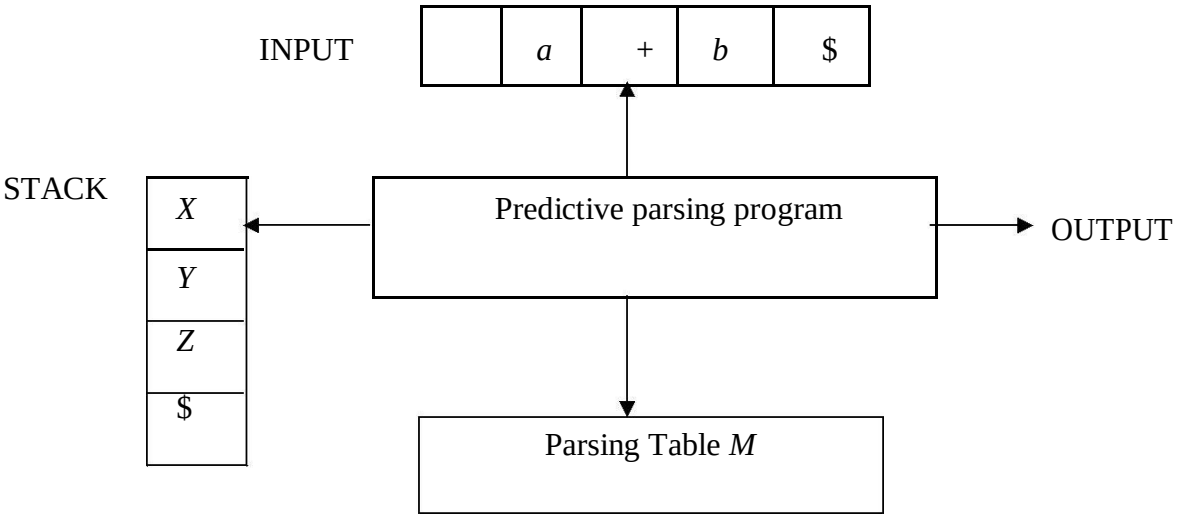
end

```
Procedure F( )
 begin
        If input -symbol='id' then ADVANCE(
        );
        else if  input-symbol='(' then ADVANCE(
        );
        E( );
        else if  input-symbol=')' then ADVANCE(
        );
end

else ERROR( );
```

2. <u>PREDICTIVE PARSING (Non Recursive Predictive Parser)</u>

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non- terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array $M[A, a]$, where **'*A*'** is anon-terminal and **'*a*'** is aterminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   If $M[X , a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by WVU. If $M[X , a]$ = **error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

Input : A string w and a parsing table M for grammar G.

Output : If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method : Initially, the parser has $\$S$ on the stack with S, the start symbol of G on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else            /* X is a non-terminal */
        if M[X, a] = X →Y1Y2 … Yk then begin
```

pop X from the stack;
push Y_k, Y_{k-1}, … ,Y_1 onto the stack, with Y_1 on top;
output the production $X \rightarrow Y_1\ Y_2 \ldots Y_k$

end
else *error*()
until X = $ /* stack is empty */

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

Rules for first( ):

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

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is non- terminal and X → $a\alpha$ is a production then add a to FIRST(X).

4. If X is non- terminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1),…,FIRST(Y_{i-1}); that is, $Y_1,\ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.

3. If ε is in FIRST(α), add A → α to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add $A \rightarrow \alpha$ to $M[A, \$]$.

4. Make each undefined entry of M be **error**.

Example:

Consider the following grammar :

E → E+T |T
T → T*F |F
F→ (E) |id

After eliminating left-recursion the grammar is

E → TE'
E' → +TE' |ε
T → FT'
T' → *FT' | ε
F → (E) |id

First( ) :

FIRST(E) ={ (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') ={*, ε }

FIRST(F) ={ (, id }

Follow( ):

FOLLOW(E) ={ $,) }

FOLLOW(E') ={ $,) }

FOLLOW(T) ={ +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) ={+, * , $,) }

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F→ id			F→ (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F→ id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F→ id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F→ id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

LL(1) grammar :

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' |a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST()and FOLLOW() for all the non-terminals.

FIRST(S) ={ i, a }

FIRST(S') = {e, ε }

FIRST(E) ={ b}

FOLLOW(S) ={ $,e }

FOLLOW(S') = { $,e }

FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S'→ eS S'→ ε			S'→ ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**. Other type is called LR Parser.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B→ d

The sentence to be recognized is **abbcde.**

Reduction (leftmost)		**Rightmost derivation**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**de	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string id_1+id_2*id_3

The rightmost derivation is :

E → **<u>E+E</u>**
→ E+**<u>E*E</u>**
→ E+E***<u>id_3</u>**
→ E+**<u>id_2</u>*** id 3
→ **<u>id_1</u>**+id_2*** id 3

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if *w* is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	$id_1+id_2*id_3$ $	shift
$ id_1	$+id_2*id_3$ $	reduce by E→id
$ E	$+id_2*id_3$ $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	$*id_3$ $	reduce by E→id
$ E+E	$*id_3$ $	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

Actions in shift -reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. **Shift-reduce conflict**: The parser cannot decide whether to shift or to reduce.

2. **Reduce-reduce conflict**: The parser cannot decide which of several reductions to make.

Viable prefixes:

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects asyntactic error as soon as possible.

Drawbacks of LR method:

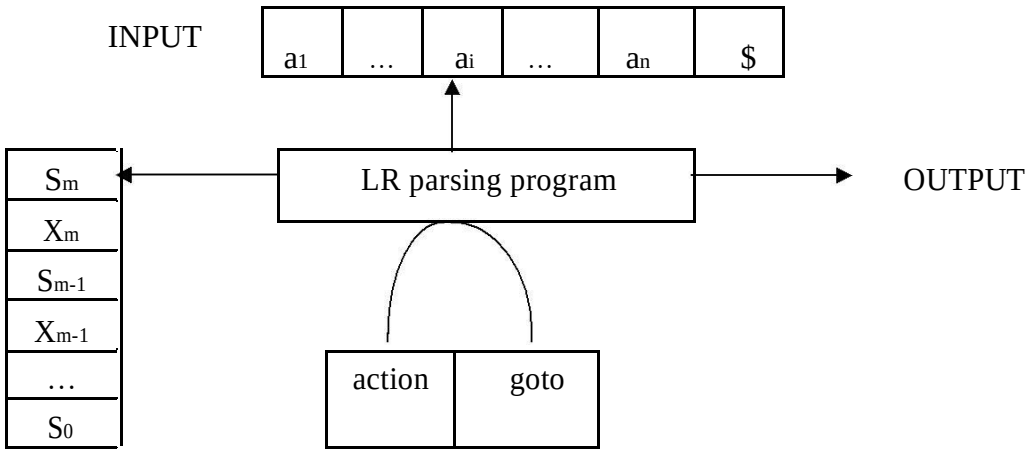
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look -Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

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

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

Closure operation:

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

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithmfor construction of SLR parsing table:

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :

1. Construct C ={$I_0, I_1, \ldots I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If[A→α·] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A)= I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F→ (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F→ (E) ------ (5)
F→ id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F→ (E)
F→ id

Step 2 : Find LR (0) items.

I_0 : E' → **.** E
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

GOTO (I_0 , E)

I_1 : E' → E **.**
E → E **.** + T

GOTO (I_4 , id)

I_5 : F→ id **.**

GOTO (I_0 , T)
I_2 : E → T .
T → T . * F

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_0 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , id)
I_5 : F→ id .

GOTO (I_1 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_2 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , E)
I_8 : F→ (E .) E
→ E . + T

GOTO (I_4 , T)
I_2 : E →T .
T → T . * F

GOTO (I_4 , F)
I_3 : T → F .

GOTO (I_6 , T)
I_9 : E → E + T .
T → T . * F

GOTO (I_6 , F)
I_3 : T → F .

GOTO (I_6 , ( )
I_4 : F→ (. E)

GOTO (I_6 , id)
I_5 : F→ id .

GOTO (I_7 , F)
I_{10} : T → T * F .

GOTO (I_7 , ( )
I_4 : F→ (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_7 , id)
I_5 : F → id .

GOTO (I_8 ,))
I_{11} : F→ (E) .

GOTO (I_8 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F→ . (E)
F→ . id

GOTO (I_9 , *)
I_7 : T → T * . F
F→ . (E)
F→ . id

GOTO (I_4 , ()

I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → id

FOLLOW (E) = { $,) , +)
FOLLOW (T) = { $, + ,) , * }
FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						GOTO		
	id	+	*	(	)	$	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F→ id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F→ id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

CANONICAL LR PARSING:

Example:

S → CC
C →cC|d.

1. Number the grammar productions:

1. S →CC
2. C →cC
3. C →d

2. The Augmented grammar is:

S^{I} →S

S →CC
C →cC
C →d.

3.Constructing the sets of LR(1) items:

We begin with:

S^{I} →.S,$ begin with look-a-head (LAH) as $. Here after with the help of closure other items are added.

Closure():

For the production A->α.Bβ,a , Function closure tells us to add [B→.r,b] for each production B→r and terminal b in FIRST (βa). Now B→r must be S→CC, and since β is ∈ and a is $, b may only be $. Thus,

S→.CC,$

We continue to compute the closure by adding all items [C→.r,b] for b in FIRST [C$] i.e., matching [S→.CC,$] against [A→α.Bβ,a] we have, A=S, α=∈, B=C and a=$. FIRST (C$) = FIRST ©

FIRST© = {c,d} We add items:

C→.cC,C

C→cC,d

C→.d,c

C→.d,d

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:

I_0 : $S^I \rightarrow .S,\$$ $S \rightarrow .CC,\$$ $C \rightarrow .CC,c/d$ $C \rightarrow .d.c/d$

Now we start computing goto (I_0,X) for various non-terminals i.e.,

Goto (I_0,S):

I_1 : $S^I \rightarrow S.,\$$ → reduced item.

Goto (I_0,C)

I_2 : $S \rightarrow C.C, \$$

$C \rightarrow .cC,\$$

$C \rightarrow .d,\$$

Goto (I_0,c) :

I_3 : $C \rightarrow c.C,c/d$

$C \rightarrow .cC,c/d$

$C \rightarrow .d,c/d$

Goto (I_0,d)

I_4 $C \rightarrow d., c/d$ → reduced item.

Goto (I_2,C)

I_5 $S \rightarrow CC.,\$$ → reduced item.

Goto (I_2,c)

I_6	C→c.C,$	
	C→.cC,$	
	C→.d,$	
Goto (I_2,d)		
I_7	C→d.,$	→ reduced item.
Goto (I_3,C)		
I_8	C→cC.,c/d	→ reduced item.
Goto (I_3,c)	I_3	
	C→c.C, c/d	
	C→.cC,c/d	
	C→.d,c/d	
Goto (I_3,d)	I_4	
	C→d.,c/d.	→ reduced item.
Goto (I_6,C)		
I_9	C→cC.,$	→ reduced item.
Goto (I_6,c)	I_6	
	C→c.C,$	
	C→,cC,$	
	C→.d,$	
Goto (I_6,d)	I_7	
	C→d.,$	→ reduced item.

All are completely reduced. So now we construct the canonical LR (1) parsing table –

Here there is no need to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

	Action			goto	
State	C	D	$	S	C
0	S3	S4		1	2

1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R4	R4			
5			R1		
6	S6	S7			9
7			R4		
8	R3	R3			
9			R3		

1. Consider I0 items:

The item S→.S.$ gives rise to goto [I0,S] = I1 so goto [0,s] = 1.

The item S→.CC, $ gives rise to goto [I0,C] = I2 so goto [0,C] = 2.

The item C→.cC, c/d gives rise to goto [I0,c] = I3 so goto [0,c] = shift 3

The item C→.d, c/d gives rise to goto [I0,d] = I4 so goto [0,d] = shift 4

2. Consider I0 items:

The item S^{I}→S.,$ is in I1, then set action [1,$] = accept

3. Consider I2 items:

The item S→C.C,$ gives rise to goto [I2,C] = I5. so goto [2,C] = 5

The item C→.cC, $ gives rise to goto [I2,c] = I6. so action [0,c] = shift 6. The item C→.d,$ gives rise to goto [I2,d] = I7. so action [2,d] = shift 7

4. Consider I3 items:

The item C→.cC, c/d gives rise to goto [I3,C] = I8. so goto [3,C] = 8

The item C→.cC, c/d gives rise to goto [I3,c] = I3. so action [3,c] = shift 3. The item C→.d, c/d gives rise to goto [I3,d] = I4. so action [3,d] = shift 4.

5. Consider I4 items:

The item C→.d, c/d is the reduced item, it is in I4 so set action [4,c/d] to reduce c→d. (production rule no.3)

6. Consider I5 items:

The item S→CC.,$ is the reduced item, it is in I5 so set action [5,$] to S→CC (production rule no.1)

7. Consider I6 items:

The item C→c.C,$ gives rise to goto [I6 ,C] = I9. so goto [6,C] = 9

The item C→.cC,$ gives rise to goto [I6 ,c] = I6. so action [6,c] = shift 6

The item C→.d,$ gives rise to goto [I6 ,d] = I7. so action [6,d] = shift 7

8. Consider I7 items:
The item C→d., $ is the reduced item, it is in I7.

So set action [7,$] to reduce C→d (production no.3)

9. Consider I8 items:

The item C→cC.,c/d in the reduced item, It is in I_8, so set action[8,c/d] to reduce C→cC (production rule no .2)

10. Consider I9 items:

The item C →cC, $ is the reduced item, It is in I9, so set action [9,$] to reduce C→cC (Production rule no.3)

If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

LALR PARSING:

Example:

1. Construct C={I0,I1,… ..... ,In} The collection of sets of LR(1) items

2. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (group them into a single term)

I_0 →same as previous

I_1 → “

I_2 → “

I_{36} –> Clubbing item I3 and I6 into one I36 item.

C →cC,c/d/$

C→cC,c/d/$
C→d,c/d/$

I_5 →same as previous

I_{47}-> Clubbing item I4 and I7 into one I47 item

C→d,c/d/$

I_{89} → Clubbing item I8 and I9 into one I89 item

C→cC, c/d/$

LALR parsing table construction:

State	Action			Goto	
	c	d			C

The Parser Generator Yacc

A translator can be constructed using Yacc in the manner illustrated in Fig. 4.55. First, a file, say `translate.y`, containing a Yacc specification of the translator is prepared. The UNIX system command

```
yacc translate.y
```

transforms the file `translate.y` into a C program called `y.tab.c` using the LALR method outlined in Algorithm 4.13. The program `y.tab.c` is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parsing table is compacted as described in Section 4.7. By compiling `y.tab.c` along with the `ly` library,

The Parser Generator Yacc

A translator can be constructed using Yacc in the manner illustrated in Fig. 4.55. First, a file, say `translate.y`, containing a Yacc specification of the translator is prepared. The UNIX system command

```
yacc translate.y
```

transforms the file `translate.y` into a C program called `y.tab.c` using the LALR method outlined in Algorithm 4.13. The program `y.tab.c` is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parsing table is compacted as described in Section 4.7. By compiling `y.tab.c` along with the `ly` library,

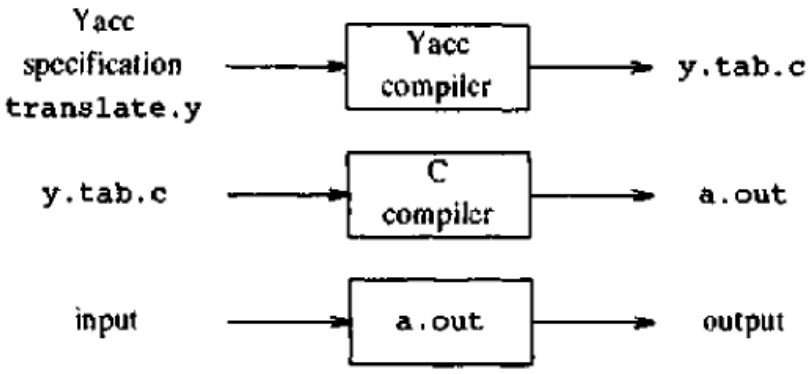


Fig. 4.55. Creating an input/output translator with Yacc.

that contains the LR parsing program using the command

```
cc y.tab.c -ly
```

we obtain the desired object program `a.out` that performs the translation specified by the original Yacc program.[6] If other procedures are needed, they can be compiled or loaded with `y.tab.c`, just as with any C program.

A Yacc source program has three parts:

```
declarations
%%
translation rules
%%
supporting C-routines
```

The declarations part. There are two optional sections in the declarations part of a Yacc program. In the first section, we put ordinary C declarations, delimited by %{ and %}. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. In

production and the associated semantic action. A set of productions that we have been writing

<left side> → <alt 1> | <alt 2> | · · · | <alt *n*>

would be written in Yacc as

```
<left side>    :   <alt 1>    { semantic action 1 }
               |   <alt 2>    { semantic action 2 }
            . . .
               |   <alt n>    { semantic action n }
               ;
```

The supporting C-routines part. The third part of a Yacc specification consists of supporting C-routines. A lexical analyzer by the name `yylex( )` must be provided. Other procedures such as error recovery routines may be added as necessary.

The lexical analyzer `yylex( )` produces pairs consisting of a token and its associated attribute value. If a token such as `DIGIT` is returned, the token must be declared in the first section of the Yacc specification. The attribute value associated with a token is communicated to the parser through a Yacc-defined variable `yylval`.

Design of a Syntax Analyzer for a Sample Language:

Designing a syntax analyzer (parser) for a sample programming language involves defining the grammar of the language and implementing the parsing algorithm. Here's a step-by-step guide to designing a syntax analyzer for a simple language, using a hypothetical "SampleLang" as an example:

Step 1: Define the Grammar

Start by defining the grammar for your SampleLang. This grammar should describe the syntax rules of the language using a formal notation like Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF). Here's a simplified example:

```
<program>      ::= <statement>+
<statement>    ::= <assignment> | <if_statement> | <while_loop>
<assignment>   ::= <identifier> '=' <expression>
<if_statement> ::= 'if' '(' <condition> ')' '{' <statement>+ '}' ['else' '{' <statement>+ '}']
<while_loop>   ::= 'while' '(' <condition> ')' '{' <statement>+ '}'
<condition>    ::= <expression> ('<' | '>' | '==' | '!=' | '<=' | '>=') <expression>
<expression>   ::= <term> ('+' | '-') <term>
<term>         ::= <factor> ('*' | '/') <factor>
<factor>       ::= <number> | <identifier> | '(' <expression> ')'
<identifier>   ::= [a-zA-Z][a-zA-Z0-9]*
<number>       ::= [0-9]+
```

This grammar defines a simple programming language with assignments, if statements, while loops, and basic arithmetic expressions.

Step 2: Choose a Parsing Algorithm

There are various parsing algorithms to choose from, such as recursive descent parsing, LL parsing, and LR parsing. The choice depends on the complexity of your language's grammar and your specific requirements. For simplicity, let's consider a recursive descent parser for this example.

Step 3: Implement the Parser

Now, you need to implement the parser based on the chosen parsing algorithm. In the case of recursive descent parsing, you'll create functions for each non-terminal symbol in your grammar.

Here's a simplified Python-like pseudocode for the parser:

```
def parse_program():
    while not end_of_input():
        parse_statement()
```


```
def parse_statement():
    if current_token() == 'if':
        parse_if_statement()
    elif current_token() == 'while':
        parse_while_loop()
    else:
        parse_assignment()

def parse_assignment():
    identifier = expect_identifier()
    expect('=')
    expression = parse_expression()

def parse_if_statement():
    expect('if')
    expect('(')
    condition = parse_condition()
    expect(')')
    expect('{')
    parse_program()
    expect('}')
    if current_token() == 'else':
        expect('else')
        expect('{')
        parse_program()
        expect('}')

def parse_while_loop():
    expect('while')
    expect('(')
    condition = parse_condition()
    expect(')')
    expect('{')
    parse_program()
    expect('}')

def parse_condition():
    left_expr = parse_expression()
    operator = expect_comparison_operator()
    right_expr = parse_expression()

def parse_expression():
```

This pseudocode represents the structure of a recursive descent parser for the SampleLang grammar. You would need to implement the actual parsing logic for each non-terminal function.

Step 4: Integrate with Lexical Analyzer

Your syntax analyzer should work in conjunction with a lexical analyzer (lexer) that tokenizes the input source code. The lexer provides the parser with a stream of tokens to parse.

Step 5: Error Handling

Implement error handling mechanisms to report syntax errors gracefully, including error messages with line and column numbers.

Step 6: Testing and Debugging

Thoroughly test your parser with various input programs to ensure it correctly recognizes valid programs and reports errors for invalid ones. Debugging tools and techniques, like printing parsing stack traces, can be invaluable.

Designing a syntax analyzer is a complex task that requires careful consideration of the language's grammar and the parsing algorithm. This example provides a high-level overview, but the actual implementation details will depend on your specific language and requirements.

Syntax Directed Definitions, Evaluation Orders for Syntax Directed Definitions, Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking.

SYNTAX – DIRECTED TRANSLATION

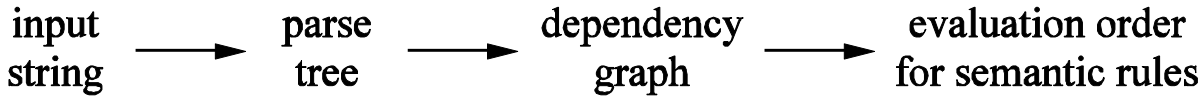
Syntax-Directed Translations

- Translation of languages guided by CFGs
- Information associated with programming language constructs
  - Attributes attached to grammar symbols
  - Values of attributes computed by "semantic rules" associated with grammar productions
- Two notations for associating semantic rules
  - Syntax-directed definitions
  - Translation schemes

Semantic Rules

- Semantic rules perform various activities:
  - Generation of code
  - Save information in a symbol table
  - Issue error messages
  - Other activities
- Output of semantic rules is the translation of the token stream

Conceptual View



- Implementations do not need to follow outline literally
- Many "special cases" can be implemented in a single pass

SYNTAX DIRECTED DEFINITIONS

Syntax directed definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called synthesized attributes and inherited attributes.

Attributes

- Each grammar symbol (node in parse tree) has attributes attached to it ex: a string, a number, a type, a memory location etc.
- Values of a Synthesized attributes at a node is computed from the values of attributes at the children of that node in the parse tree.
- Values of a Inherited attributes at a node is computed from the values of attributes at the siblings and parent of that node.
  A dependency graph represents dependencies between attributes
  A parse tree showing the values of attributes at each node is an **annotated parse tree**
- Each semantic rule for production A -> α has the form
  $b := f(c_1, c_2, \ldots, c_k)$
  - f is a function
  - b may be a synthesized attribute of A or

- b may be an inherited attribute of one of the grammar symbol on the right side of the production
- $c_1, c_2, \ldots, c_k$ are attributes belonging to grammar symbols of production

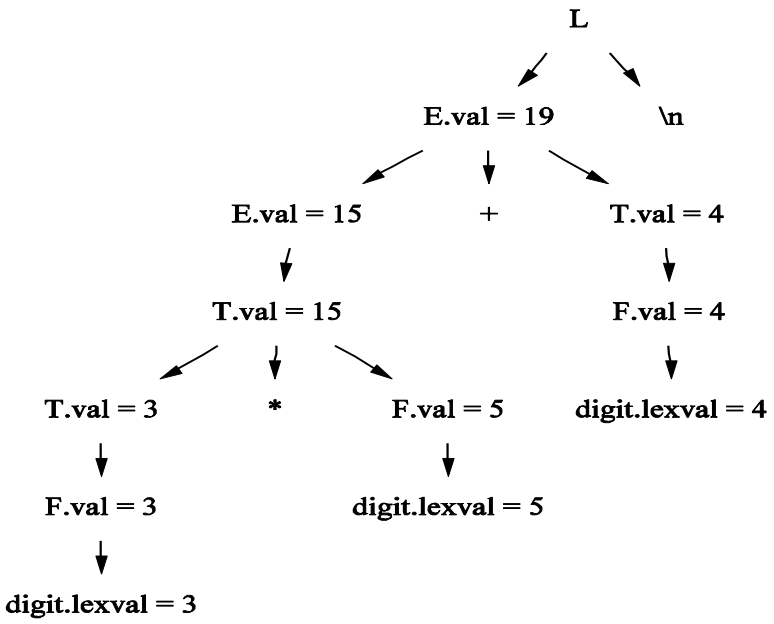
- An attribute grammar is one in which the functions in semantic rule cannot have side effects
  NOTE: a semantic rule may have side effects ex: printing a value or updating a global variable.

S-attributed Definitions

- Synthesized attributes are used extensively in practice
- S-attributed definition: A syntax-directed definition using only synthesized attributes
- Parse tree can be annotated by evaluation nodes during a single **bottom up pass**
  S-attributed Definition Example
  Desk calculator

Production	Semantic Rules
L → E n	print(E.val)
E → E_1 + T	E.val := E_1.val + T.val
E → T	E.val := T.val
T → T_1 * F	T.val := T_1.val * F.val
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digit	F.val := digit.lexval

Annotated Parse Tree Example



NOTE

In a syntax directed definations, terminals are assumed to have Synthesized attributes only,as the definations does not provide any semantic rules for terminals.values for attributes of terminals are usually supplied by the lexical analyser.Start symbol is assumed not to have any inherited attribute otherwise stated.

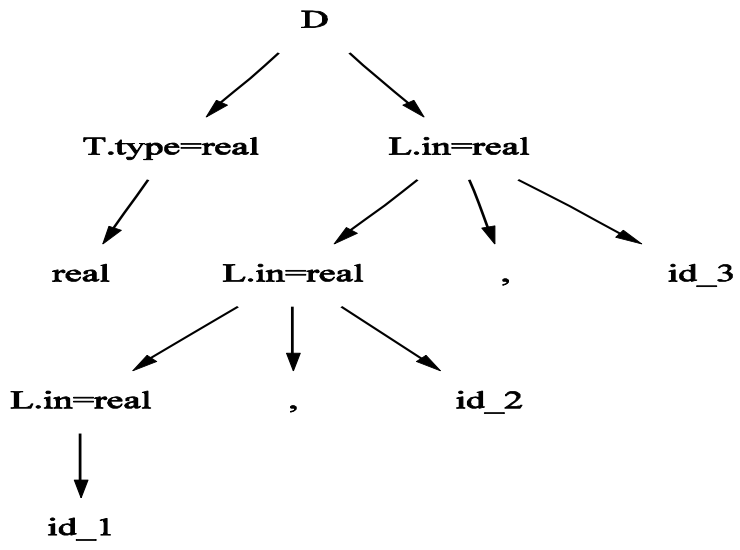
Inherited Attributes

- Inherited Attributes:
  - Value at a node in a parse tree depends on attributes of parent and/or siblings
  - Convenient for expressing dependencies of programming language constructs on context
- It is always possible to avoid inherited attributes, but they are often convenient

Inherited Attributes Example

Production	Semantic Rules
D → T L	L.in := T.type
T → int	T.type := integer
T → real	T.type := real
$L \rightarrow L_1$, id	L_1.in := L.in addtype(id.entry, L.in)
L → id	addtype(id.entry, L.in)

Annotated Inherited Attributes



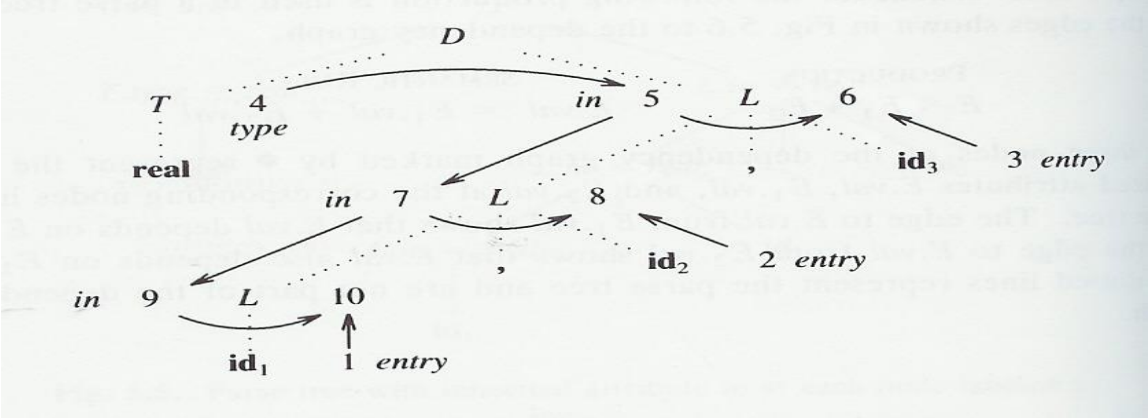
Dependency Graphs

- Dependency graph:
  - Depicts interdependencies among synthesized and inherited attributes
  - Includes dummy nodes for procedure calls
- Numbered with a topological sort
  - If $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering
  - Gives valid order to evaluate semantic rules

Creating a Dependency Graph

for each node *n* in parse tree
  for each attribute *a* of grammar symbol at *n*
    construct a node in dependency graph for *a*
for each node n in parse tree
  for each semantic rule $b := f(c_1, c_2, \ldots, c_k)$
      associated with production used at *n*
    for i := 1 to k
      construct edge from node for c_i to node for *b*

Example(inherited attribute)



Two sub-classes of the syntax-directed definitions:

- **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
- **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

To implement S-Attributed Definitions and L-Attributed Definitions we can evaluate semantic rules in a single pass during the parsing.

Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

BOTTOM-UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

- We put the values of the synthesized attributes of the grammar symbols into a parallel stack.
  - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- We evaluate the values of the attributes during reductions.

Bottom-Up Evaluation Example

Production	Code Fragment
(1) L → E \n	Print(val[top])
(2) E → E_1 + t	val[ntop] := val[top-2] + val[top]
(3) E → T	
(4) T → `T_1 * F	val[ntop] := val[top-2] * val[top]
(5) T → F	
(6) F → (E)	val[ntop] := val[top-1]
(7) F → digit	

Bottom-Up Evaluation Example

Input	State	Val	Rule
3*5+4\n	---	---	
*5+4\n	3	3	
*5+4\n	F	3	(7)
*5+4\n	T	3	(5)
5+4\n	T*	3_	
+4\n	T*5	3_5	
+4\n	T*F	3_5	(7)
+4\n	T	3_5	(4)
+4\n	E	15	(3)
4\n	E+	15_	
\n	E+4	15_4	
\n	E+F	15_4	(7)
\n	E+T	15_4	(5)
\n	E	19	(2)

	E\n	19_	
	L	19	(1)

TOP DOWN EVALUATION OF L-ATTRIBUTED DEFINITION

A top-down parser can evaluate attributes as it parses if the attribute values can be computed in a top-down fashion. Such attribute grammars are termed *L-Attributed*. First, we introduce a new type of symbol called an *action* symbol. Action symbols appear in the grammar in any place a terminal or nonterminal may appear. They may also have their own attributes. They may, however, be pushed onto their own stack, called a semantic stack or attribute stack.

We illustrate action symbols using the notation "<>" which indicates that the symbol within the brackets is to be pushed onto the semantic stack when it appears at the top of the parse stack. By inserting this action in appropriate places, we will create a translator which converts from infix expressions to postfix expressions.

EXAMPLE 6 Converting from infix to postfix via an action symbol

```
E → TE'
E → T
E' → + T E' <+>
        | ϵ
T → F T'
T → F
T' → * F T' <*>
        | ϵ
F → (E)
      | Lit   <Lit>
      | Id    <Id>
```

We parse and translate *a* + *b* * *c*. The top is on the left for both stacks.

Parse Stack	Input	Semantic Stack
E $	a + b * c $	
T E' $	a + b * c $	
F E' $	a + b * c $	
a <a> E' $	a + b * c $	
E' $	+ b * c $	
E' $	+ b * c $	<a>
+ T E' <+> $	+ b * c $	<a>
T E' <+> $	b * c $	<a>
F T' E' <+> $	b * c $	<a>
b <b> T' E' <+> $	b * c $	<a>
T' E' <+> $	* c $	<b> <a>
* F T' <*> E' <+> $	* c $	<b> <a>
F T' <*> E' <+> $	c $	<b> <a>
c <c> T' <*> E' <+> $	c $	<b> <a>
T' <*> E' <+> $	$	<c> <b> <a>
ϵ <*> E' <+> $	$	<c> <b> <a>
<*> E' <+> $	$	<c> <b> <a>
E' <+> $	$	<*> <c> <b> <a>
ϵ <+> $	$	<*> <c> <b> <a>
<+> $	$	<*> <c> <b> <a>
$	$	<+> <*> <c> <b> <a>

When the semantic stack is popped, the translated string is:

a b c * +

the input string translated to postfix. In Example 6, the action symbol did not have any attached attributes.

The BNF in Example 6 is in LL(1) form. This is necessary for the top-down parse.

The formal definition of an L-attributed grammars is as follows. An attribute grammar is *L-attributed* if and only if for each production $X_0 \rightarrow X_1 X_2 \ldots X_i \ldots X_n$,

(1) $\{X_i.inh\} = f(\{X_j.inh\}, \{X_k.att\})$ i, j >= 1, 0<=k<i

(2) $\{X_0.syn\} = f(\{X_0.inh\}, \{X_j.att\})$ 1<=j<=n

(3) {ActionSymbol.*Syn*} = f ({ActionSymbol.*Inh*})

(1) says that each ***inherited*** attribute of a symbol on the right-hand side depends only on inherited attributes of the right-hand side and arbitrary attributes of the symbols to the *left* of the given right-hand side symbol.

(2) says that each ***synthesized*** attributes of the left-hand-side symbol depends only on inherited attributes of that symbol and arbitrary attributes of right-hand-side symbols.

(2) says that the **synthesized** attributes of any action symbol depend only on the inherited attributes of the action symbol.

Conditions (1), (2), and (3) allow attributes to be evaluated in one left-to-right pass

If the underlying grammar is LL(1), then an L-attributed grammar allows attributes to be evaluated while parsing. The evaluation algorithm is:

Algorithm
LL(1) L-Attributed Evaluation

```
FOR each predicted production X0 → X1X2...Xn.
 Push X0's inherited attributes onto semantic stack.
 Push X1's inherited attributes onto semantic stack.
 Parse X1, then push X1's synthesized attributes onto
    semantic stack.
 Push X2's inherited attributes onto semantic stack.
 Parse X2, then push X2's synthesized attributes onto
    semantic stack ...
 Push Xn's inherited attributes onto semantic stack.
 Parse Xn, then push Xn's synthesized attributes onto
    semantic stack.
 Pop attributes of X1X2,...,Xn.
 Push synthesized attributes of X0.
 ENDFOR
```

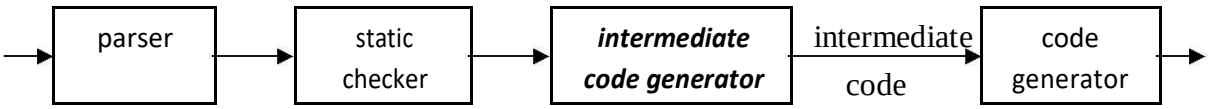
INTRODUCTION – Intermediate Code generator

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

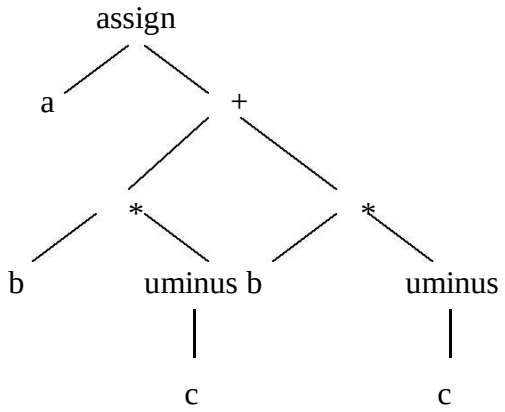
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

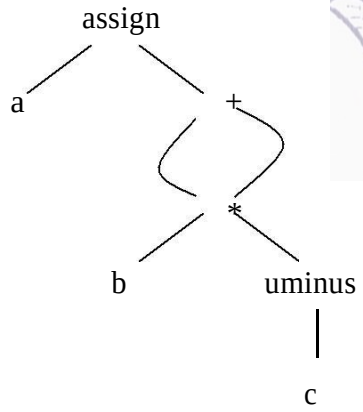
Graphical Representations:

SYNTAX TREE:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition for construction of Syntax Tree:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

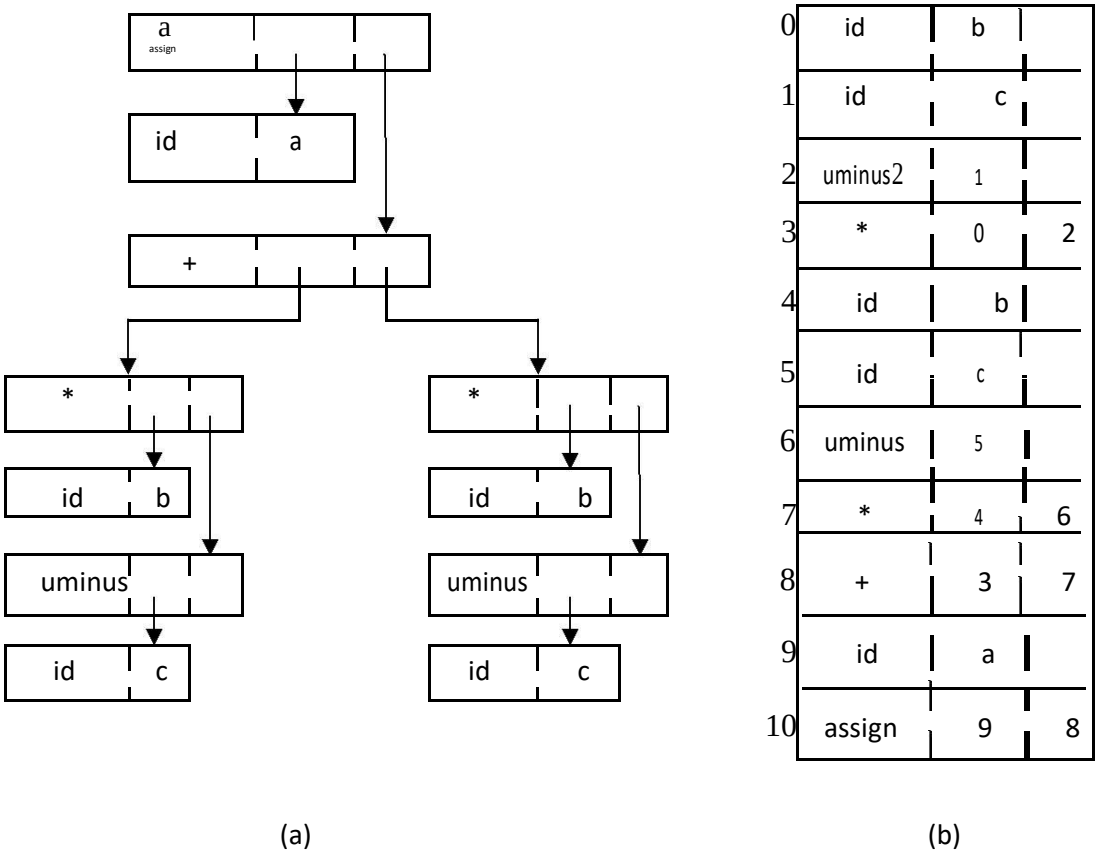
PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 +E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



(a) (b)

THREE-ADDRESS CODE:

Three-address code is a sequence of statements of the general form

x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into asequence

t_1 : = y * z

t_2 : = x + t_1

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

(a) Code for the syntax tree

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

(b) Code for the dag

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three -Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation

relop to y. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,

```
param x1
param x2
  . . .
param xn
call p,n
```

generated as part of a call of the procedure p(x_1, x_2, …. ,x_n).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : = *E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

Given input a : = b * - c + b * - c, the three-address code is as shown above.
The synthesized attribute *S.code* represents the three-address code for the assignment *S.*
The nonterminal *E* has two attributes :
1. *E.place*, the name that will hold the value of E , and
2. *E.code*, the sequence of three-address statements evaluating E.

Syntax -directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
S → id : = E	***S.code : = E.code || gen(id.place ':=' E.place)***
E → E1 + E2	***E.place := newtemp;*** ***E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place)***
E → E1 * E2	***E.place := newtemp;*** ***E.code := E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)***
E → - E1	***E.place := newtemp;*** ***E.code := E1.code || gen(E.place ':=' 'uminus' E1.place)***
E → (E1)	***E.place : = E1.place;*** ***E.code : = E1.code***
E → id	***E.place : = id.place;*** ***E.code : = ' '***

- The function *newtemp* returns a sequence of distinct names $t_1, t_2, \ldots$ in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x, y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S*** → **while** ***E*** **do** ***S1*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

Write three address code for the below expression.

a=b+c*d-e/f

t1=c*d

t2=e/f

t3=b+t1

t4=t3-t2

a=t4

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : = y op z** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2.*
- The fields *arg1* and *arg2,* for the arguments of *op,* are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples.*

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id : *T.***
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset,* and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter(name, type, offset*) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors pointer and array. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

P → D	***{ offset : = 0 }***
D → D ; D	
D → id : T	***{ enter(id.name, T.type, offset);*** ***offset : = offset + T.width }***
T → integer	***{ T.type : = integer;*** ***T.width : = 4 }***
T → real	***{ T.type : = real;*** ***T.width : = 8 }***
T → array [num] of T_1	***{ T.type : = array(num.val, T_1.type);*** ***T.width : = num.val X T_1.width }***
T → ↑ T_1	***{ T.type : = pointer (T_1.type);*** ***T.width : = 4 }***

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

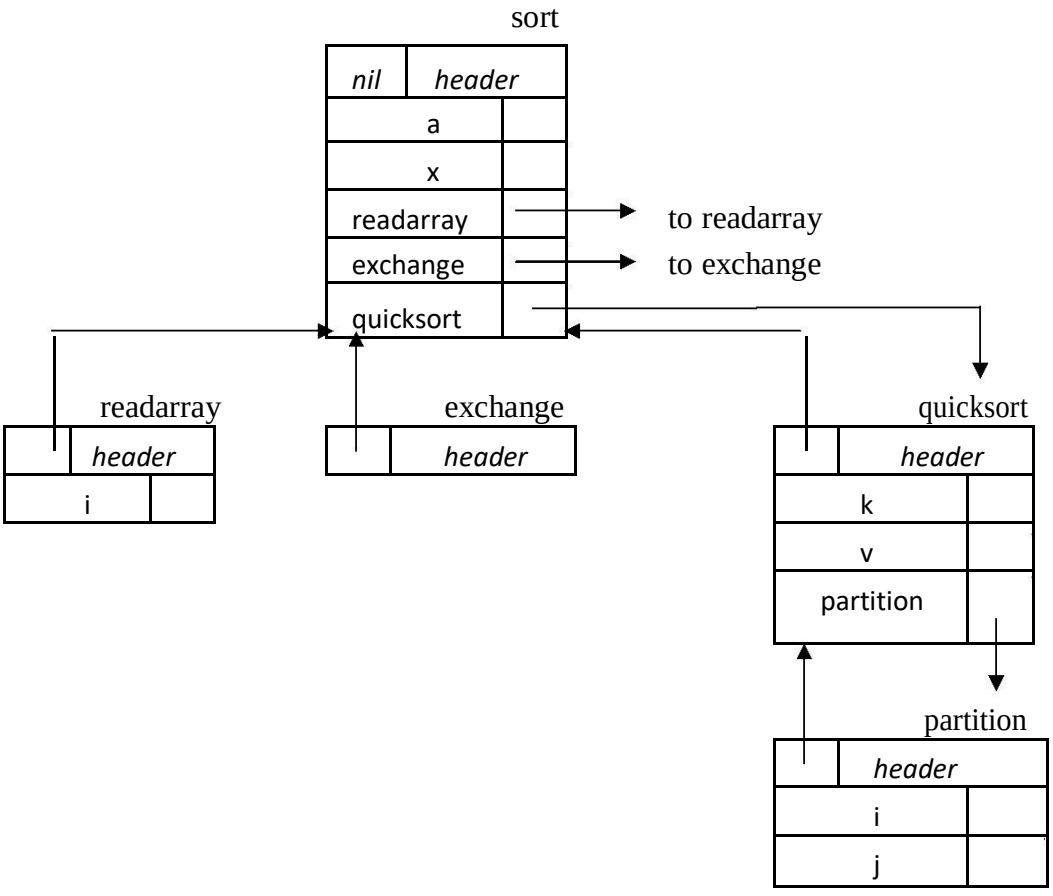
$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ D\ ;\ S$

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow \textbf{proc id}\ D_1;S$ is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray*, exchange, and *quicksort* pointing back to that for the containing procedure sort, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

P → M D	***{ addwidth (top(tblptr) , top (offset));*** ***pop (tblptr); pop (offset) }***
M → ε	***{ t : = mktable (nil);*** ***push (t,tblptr); push (0,offset) }***
D → D1 ; D2	
D → proc id ; N D1 ; S	***{ t : = top (tblptr);*** ***addwidth (t, top (offset));*** ***pop (tblptr); pop (offset);*** ***enterproc (top (tblptr), id.name, t) }***
D → id : T	***{ enter (top (tblptr), id.name, T.type, top (offset));*** ***top (offset) := top (offset) + T.width }***
N → ε	***{ t := mktable (top (tblptr));*** ***push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.
- The top element of stack *offset* is the next available relative address for a local of the current procedure.
- All semantic actions in the subtrees for B and C in

  A → BC {*action*$_A$}

  are done before *action*$_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil)*. The action also pushes relative address 0 onto stack offset.
- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.
- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.
- When the action on the right side of $D \rightarrow proc\ id;\ ND_1;\ S$ occurs, the width of all declarations generated by D_1 is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **relop** E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E → E **or** E | E **and** E | **not** E | (E) | **id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for
  a **or** b **and not** c is the three-address sequence

t_1 : = **not** c t_2

: = b **and** t_1 t_3

: = a **or** t_2

- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

100 : if a < b goto 103
101 : t : = 0
102 : goto 104
103 : t : = 1
104 :

Translation scheme using a numerical representation for booleans

$E \rightarrow E_1$ **or** E_2	*{ E.place : = newtemp;*
	*emit(E.place ': =' E_1.place '***or***'E_2.place)}*
$E \rightarrow E_1$ **and** E_2	*{ E.place : = newtemp;*
	*emit(E.place ': =' E_1.place '***and***'E_2.place)}*
$E \rightarrow$ **not** E_1	*{ E.place : = newtemp;*
	*emit(E.place ': =' '***not***' E_1.place)}*
$E \rightarrow (E_1)$	*{ E.place : = E_1.place }*
$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	*{ E.place : = newtemp;*
	emit('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* '**goto**' *nextstat* + **3***);*
	*emit(E.place ': =' '***0***');*
	emit('**goto**' *nextstat* +**2***);*
	*emit(E.place ': =' '***1***') }*
$E \rightarrow$ **true**	*{ E.place : = newtemp;*
	*emit(E.place ': =' '***1***') }*
$E \rightarrow$ **false**	*{ E.place : = newtemp;*
	*emit(E.place ': =' '***0***') }*

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and, or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto 103	107 : t_2 : = 1
101 : t_1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t_3 : = 0
103 : t_1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t_3 : = 1
105 : t_2 : = 0	112 : t_4 : = t_2 and t_3
106 : goto 108	113 : t_5 : = t_1 or t_4

Control-Flow Translation of Boolean Expressions:

With the help of control flow mechanism, the Boolean operator and conditional statements in which Boolean expression are part of it are translated into three address code as follows.

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	*E_1.true : = E.true;* *E_1.false : = newlabel;* *E_2.true : = E.true;* *E_2.false : = E.false;* *E.code : = E_1.code || gen(E_1.false ':') || E_2.code*
$E \rightarrow E_1$ **and** E_2	*E_1.true : = newlabel;* *E_1.false : = E.false;* *E_2.true : = E.true;* *E_2.false : = E.false;* *E.code : = E_1.code || gen(E_1.true ':') || E_2.code*
$E \rightarrow$ **not** E_1	*E_1.true : = E.false;* *E_1.false : = E.true;* *E.code : = E_1.code*

$E \rightarrow (E1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow$ **id**$_1$ **relop id**$_2$	$E.code := gen($'**if**' **id**$_1$.*place* **relop.***op* **id**$_2$.*place* '**goto**' *E.true)* || *gen*('**goto**' *E.false*)
$E \rightarrow$ **true**	*E.code := gen(*'**goto**' *E.true)*
$E \rightarrow$ **false**	*E.code := gen('***goto***' E.false)*

Flow-of-Control Statements

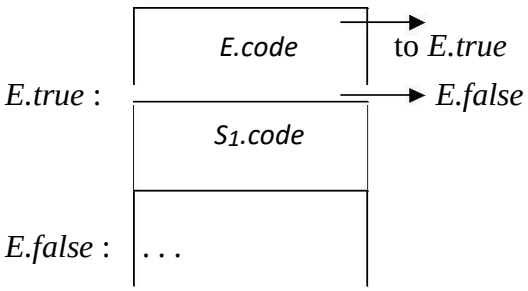
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

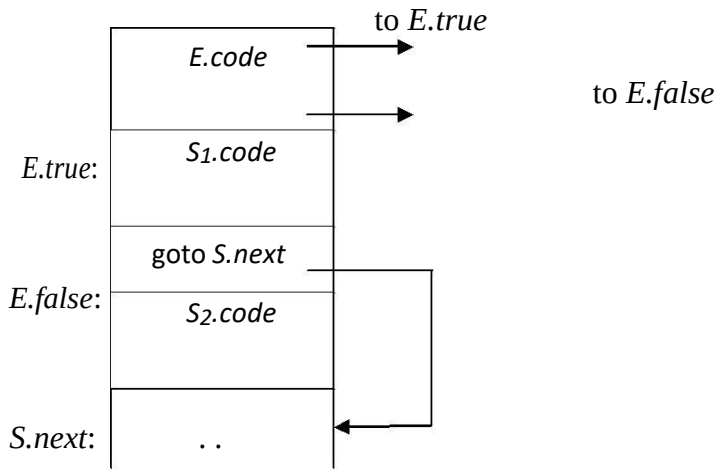
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

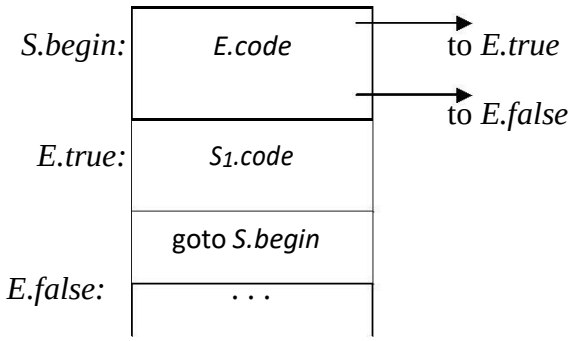
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then -else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	*E.true : = newlabel;* *E.false : = S.next;* *S_1.next : = S.next;* *S.code : = E.code || gen(E.true ':') || S_1.code*
$S \rightarrow$ **if** E **then** S_1 **else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* *S_1.next : = S.next;* *S_2.next : = S.next;* *S.code : = E.code || gen(E.true ':') || S_1.code ||* *gen('**goto** ' S.next) ||* *gen(E.false ':') || S_2.code*
$S \rightarrow$ **while** E **do** S_1	*S.begin : = newlabel;* *E.true : = newlabel;* *E.false : = S.next;* *S_1.next : = S.begin;* *S.code : = gen(S.begin ':')|| E.code ||* *gen(E.true ':') || S_1.code ||* *gen('**goto** ' S.begin)*

TYPE CHECKING

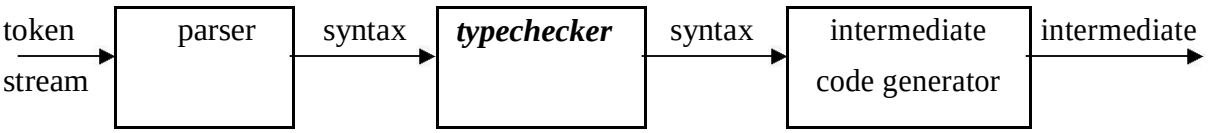
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking,* detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting

"the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.
Constructors include:
Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T_1 and T_2 are type expressions, then their Cartesian product T_1 X T_2 is a type expression.
Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.
For example:

```
type row = record
             address: integer;
             lexeme: array[1..15] of char
           end;
var table: array[1...101] of row;
```

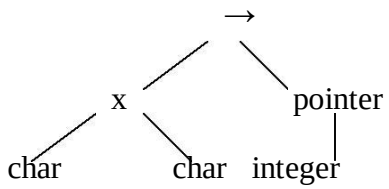
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".
For example, ***var p: ↑ row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → *pointer* (integer)



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checkingdone by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E
D → D ; D
D → id : T { *addtype* (id.*entry* , T.*type*)}
T → char { T.*type* : = char }
T → integer { T.*type* : = integer }
T → ↑ T1 { T.*type* : = pointer(T_1.*type*) }
T → array [num] of T1 { T.*type* : = array (1… num.val , T_1.*type*) }

In the above language,

→ There are two basic types : char and integer ;

→ *type_error* is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* forE gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E_1 **mod** E_2 { *E.type* : = **if** E_1. *type* = *integer* **and**
   E_2. *type* = *integer* **then** *integer*
   else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E_1 [E_2] { *E.type* : = if E_2.*type* = *integer* **and**
   E_1.*type* = *array(s,t)* **then** *t*
   else *type_error* }
   In an array reference E_1 [E_2] , the index expression E_2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

5. E → E_1 ↑ { *E.type* : = **if** E_1.*type* = *pointer (t)* **then** *t*
   else *type_error* }

   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if** **id**.*type* = E.*type* **then** *void* **else**
type_error }

2. Conditional statement:

S → if E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

S → S_1 ; S_2 { S.*type* : = **if** S_1.*type* = *void* and S_2.*type* = *void*
then *void*
else *type_error* }

Type checking of functions

The rule for checking the type of a function application is :

E → E_1 (E_2) { E.*type* : = **if** E_2.*type* = *s* **and**
E_1.*type* = *s* → *t* **then** *t*
else *type_error* }

Evaluation of S-Attribute Definitions:

S-attributes, also known as synthesised attributes, are a concept used in compiler design and parsing theory to attach information to the nodes of a syntax tree or abstract syntax tree (AST) during parsing. These attributes are computed during a bottom-up parsing process and are typically used for tasks like type checking, code generation, and semantic analysis. The evaluation of S-attributes is an essential part of the semantic analysis phase in a compiler. Here's an evaluation of S-attribute definitions:

Advantages of S-Attributes:

1. **Simplified Semantic Analysis:** S-attributes provide a straightforward way to associate semantic information with the nodes of the syntax tree. This simplifies the process of performing semantic checks and transformations.

2. **Ease of Use:** S-attributes are relatively easy to understand and implement. They follow a predictable flow from the bottom of the tree to the top, allowing for clear and modular definitions.

3. **Efficiency:** S-attribute evaluation can be performed efficiently during the parsing process, without the need for a separate pass over the AST. This can save memory and processing time.

4. **Integration with Parsing:** S-attributes are well-suited for integration with bottom-up parsing techniques like LR parsing. They complement the parsing process by performing semantic actions as the parsing algorithm proceeds.

Limitations and Considerations:

1. **Limited Expressiveness:** S-attributes are primarily used for simple tasks like type checking and basic semantic analysis. They may not be expressive enough for more complex transformations or optimizations.

2. **Order Dependencies:** The order of S-attribute evaluations matters, as they depend on attributes computed for child nodes. Care must be taken to ensure that the evaluation order is well-defined and consistent.

3. **Context Sensitivity:** S-attributes may struggle with context-sensitive analysis, where semantic decisions depend on a broader context than just the parent and child nodes. In such cases, additional techniques like symbol tables and context-dependent attributes may be necessary.

4. **Potential for Code Duplication:** In larger grammars or complex languages, S-attribute definitions can become verbose, leading to potential code duplication or maintenance challenges.

5. **Error Handling:** Handling errors during S-attribute evaluation can be complex. Proper error reporting and recovery mechanisms should be in place to provide meaningful feedback to the user.

6. **Limited Support for Optimization:** S-attributes are primarily used for semantic analysis and may not be suitable for advanced compiler optimizations. For optimization, you may need additional structures or techniques.

In summary, S-attributes are a valuable tool for attaching semantic information to nodes in a syntax tree during parsing. They simplify many aspects of semantic analysis and integrate well with bottom-up parsing techniques. However, their expressiveness is limited, and they may not be suitable for all aspects of compiler design, especially for complex languages or advanced optimizations. Compiler designers should carefully consider their use in the context of the specific language and requirements of the compiler being developed.

Design of Predictive Translator:

A predictive translator is a type of compiler or translator that converts source code from one programming language to another without necessarily preserving the original structure or semantics. This kind of translator is often used for tasks like code migration, where you want to move code from an old language to a new one. Here's a step-by-step guide to designing a predictive translator:

Step 1: Define the Source and Target Languages

Start by defining the source and target languages for your translator. Understand the syntax, semantics, and features of both languages. It's essential to have a clear understanding of what constructs in the source language need to be mapped to in the target language.

Step 2: Lexical Analysis (Scanning)

Implement a lexer (lexical analyzer) for the source language. The lexer breaks down the source code into tokens. This is a necessary step to understand the structure of the source code.

Step 3: Syntactic Analysis (Parsing)

Create a parser for the source language. The parser should generate a parse tree or abstract syntax tree (AST) that represents the structure of the source code. If the source language has a well-defined grammar, use a parser generator like ANTLR or yacc to create the parser.

Step 4: Define Translation Rules

Define a set of translation rules or patterns that specify how each construct in the source language should be translated to the target language. This step is crucial as it forms the core of your translator.

For example, if you're translating code from a procedural language to an object-oriented language, you'll need rules to map procedures to classes and functions, variables to attributes, and so on.

Step 5: Tree Traversal and Translation

Write a tree traversal algorithm that traverses the parse tree or AST generated by the parser. During traversal, apply the translation rules defined in the previous step to generate equivalent code in the target language.

Here's a simplified pseudocode for tree traversal:

```
function translate(node):
    if node is a procedure_declaration:
        generate_class(node)
    elif node is a variable_declaration:
        generate_attribute(node)
    elif node is a function_declaration:
        generate_method(node)
    # Handle other constructs and recursively traverse the tree.
    for child in node.children:
        translate(child)
```

Step 6: Code Generation

Implement the code generation phase. This phase involves taking the translated AST and emitting code in the target language. Ensure that the generated code adheres to the target language's syntax and conventions.

Step 7: Error Handling

Include error-handling mechanisms to report and handle translation errors. Errors can occur when the source code cannot be accurately translated into the target language due to syntactic or semantic differences.

Step 8: Testing and Validation

Test your predictive translator with a variety of source code examples to ensure it produces correct and valid code in the target language. Validation against a suite of test cases is crucial for ensuring the accuracy of the translation.

Step 9: Documentation and Usage

Provide documentation on how to use the predictive translator. Explain any limitations and assumptions made during the translation process.

Step 10: Optimization (Optional)

Consider adding optimization steps to improve the quality of the translated code. These may include simplification of expressions, removal of dead code, or other optimizations that make the generated code more efficient.

Remember that designing a predictive translator can be a complex task, particularly when translating between languages with significant differences in syntax and semantics. Thoroughly understanding both the source and target languages and carefully defining translation rules are essential for success.

Specification of Simple Type Checker:

A simple type checker is a component of a compiler or interpreter that verifies the correctness of types in a program according to the language's type system. It helps catch type-related errors, ensuring that operations are performed on compatible data types and that variables are used in a manner consistent with their declarations. Here's a specification for a simple type checker:

1. Input: The input to the type checker is typically the abstract syntax tree (AST) or another representation of the source code generated by the parser.

2. Output: The type checker may output error messages or warnings when it encounters type-related issues in the code. It may also annotate the AST or generate additional data structures to store type information.

3. Type System: Define the type system of the programming language you are checking. This includes specifying the basic data types (e.g., int, float, string, boolean) and the rules for type compatibility, type promotion, and type coercion.

4. Type Annotations: If your AST nodes do not already contain type information, introduce type annotations for variables, expressions, function parameters, and other relevant program elements. These annotations help the type checker keep track of types during analysis.

5. Symbol Table: Maintain a symbol table to store information about declared variables, their types, and their scopes. The symbol table is essential for looking up variable types during type checking.

6. Type Checking Rules:

a. **Variable Declarations:** Check that variable declarations are consistent with the types assigned to them. For example, ensure that the declared type of a variable matches the type of the expression used to initialize it, or vice versa.

b. **Type Compatibility:** Enforce type compatibility rules for operations, assignments, and function calls. For instance, addition should be allowed for numeric types, but not for a string and an integer.

c. **Type Coercion:** Implement rules for type coercion or implicit type conversion when the language allows it. For instance, if a language permits converting an integer to a float automatically, the type checker should handle this.

d. **Function Calls:** Verify that function calls match the types and number of arguments expected by the function's declaration.

e. **Expression Evaluation:** Ensure that expressions are well-typed. For binary operations, check that the types of operands are compatible.

f. **Type Inference (Optional):** Implement type inference if your language supports it. Type inference allows the type checker to deduce types for variables or expressions without explicit type annotations.

7. Error Handling: When a type error is encountered, report an error message that includes the location (line and column number) of the error and a description of the issue. Optionally, you can include suggestions for fixing the error.

8. Recursion and Scoping: Implement recursive type checking to handle nested scopes and function calls. Ensure that variable scope and visibility rules are enforced.

9. Testing: Test the type checker with a variety of input programs, including both valid and invalid code. Create test cases that cover different aspects of the type system, such as type coercion, type inference, and complex expressions.

10. Integration: Integrate the type checker into the compiler or interpreter pipeline, typically after parsing and before code generation or execution.

11. Documentation: Provide documentation for users and developers on how to use and extend the type checker. Include details on the supported type system and any language-specific type checking rules.

Building a type checker is a fundamental component of a compiler or interpreter and requires a deep understanding of the language's type system and syntax. It plays a crucial role in ensuring program correctness and robustness.

Back Patching:

Backpatching is a technique used in compiler design and code generation to handle jumps or branches in a program. It's particularly useful when generating code for control flow constructs like if statements, loops, and goto statements. The purpose of backpatching is to fill in the target addresses or labels of these control flow instructions after their targets are known, typically during later stages of code generation. Here's how backpatching works:

Step 1: Identify Placeholder Locations

During the initial code generation phase, when the compiler encounters control flow instructions with unknown target addresses or labels, it doesn't generate the final address or label at that point. Instead, it leaves a placeholder or a marker that signifies an unresolved jump target.

For example, in a hypothetical assembly-like language, when generating code for an `if` statement:

```
if condition then
   jump_to_????
endif
```

The "jump_to_????" part represents a placeholder for the target address. The compiler doesn't yet know where the "endif" label will be placed in the generated code.

Step 2: Record Placeholder Locations

While generating code, the compiler keeps track of the locations of these placeholders and the control flow instructions that reference them. Typically, it maintains a list or data structure that records the locations where placeholders were inserted and associates them with the corresponding control flow instructions.

Step 3: Resolve Target Addresses

Once the compiler has generated the entire code and knows the final locations of labels or targets (e.g., after processing all statements and control structures), it goes back to the recorded placeholder locations.

At this point, the compiler can fill in the actual target addresses or labels at the placeholder locations. This process is called backpatching. The compiler replaces the placeholder with the correct target information.

Step 4: Generated Code with Resolved Targets

After backpatching, the generated code now looks like this:


```
if condition then
   jump_to_1234  ; "1234" is the actual address of "endif"
endif
```

The target address has been resolved, and the generated code is now complete and can be executed correctly.

Use Cases for Backpatching:

1. **If Statements:** Backpatching is commonly used when generating code for if-else statements. It's used to update the jump targets for both the "if" and "else" branches after the locations of the "endif" labels are known.

2. **Loops:** For loop constructs like while and for loops, backpatching is used to update the jump targets for looping conditions and exit points.

3. **Goto Statements:** In languages that support goto statements, backpatching helps associate labels with the actual locations in the generated code.

4. **Switch Statements:** Backpatching can also be applied in switch statements, where it's used to resolve the jump targets for different case labels.

Backpatching is an essential technique in code generation, ensuring that control flow instructions are correctly directed to their targets in the final code. It simplifies the code generation process and allows for efficient handling of complex control structures.

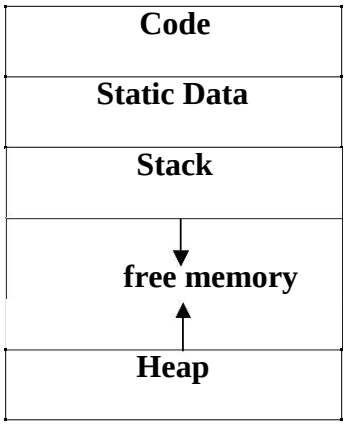
UNIT IV RUN-TIME ENVIRONMENT AND CODE GENERATION 8

Storage Organization, Stack Allocation Space, Access to Non-local Data on the Stack, Heap Management - Issues in Code Generation - Design of a simple Code Generator.

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation

tree at the bottom, the latter activation has its record at the top of the stack.

- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

Temporaries
Local Data
Machine Status
Control Link
Access Link
Actual Parameters
Return Value

- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.
- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

Static allocation

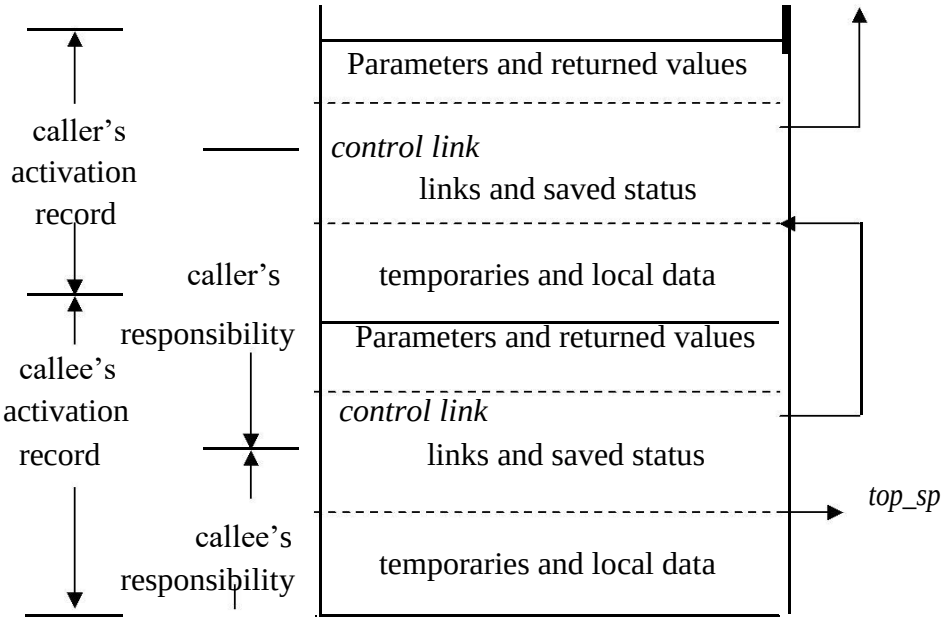
- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

Stack allocation

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.



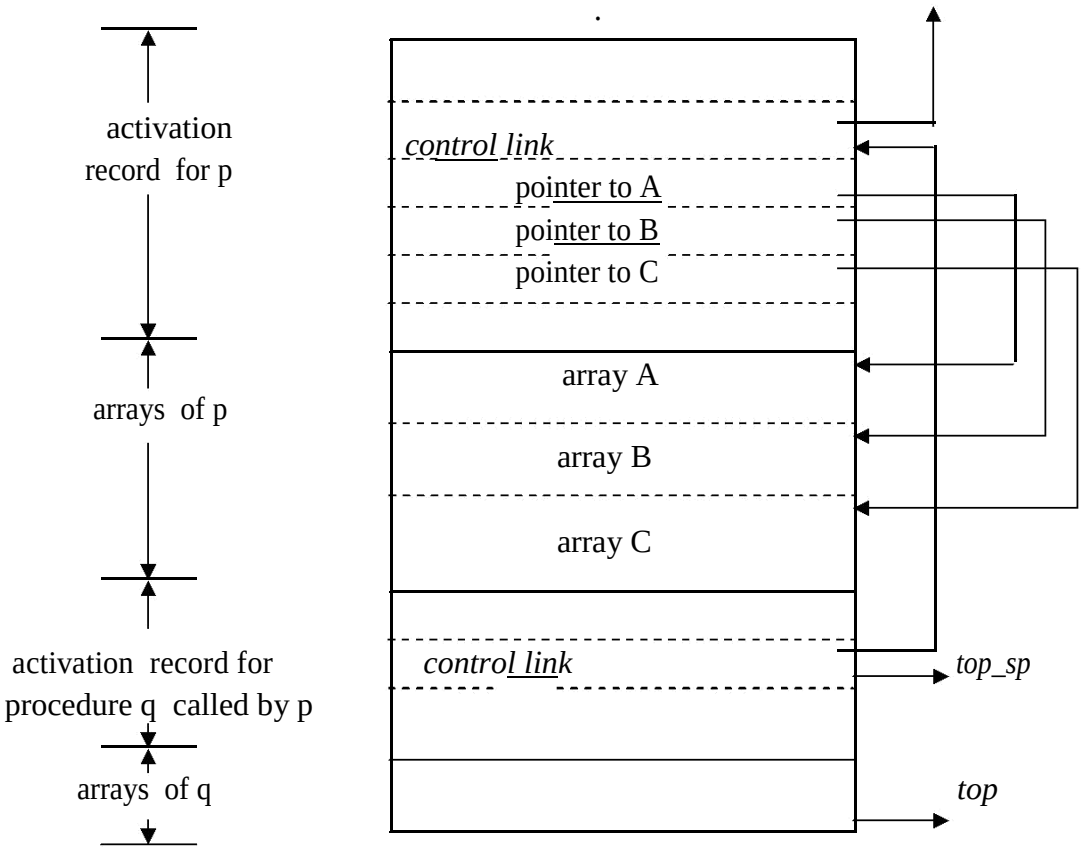
Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.

- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

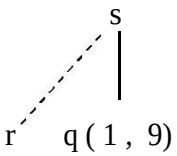
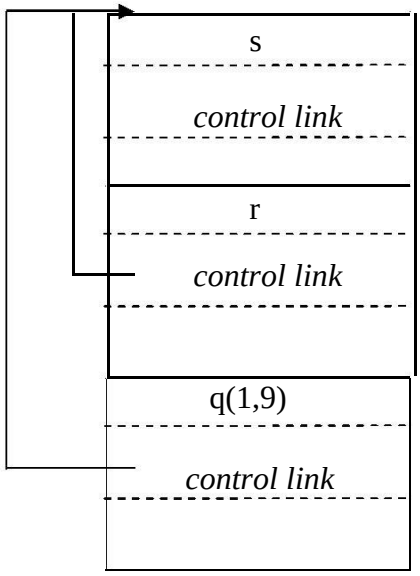
- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

Heap allocation

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management

4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

- Intermediate representation can be :
  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
     - It can be placed in a fixed memory location and can be executed immediately.

  b. Relocatable machine language
     - It allows subprograms to be compiled separately.

  c. Assembly language
     - Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

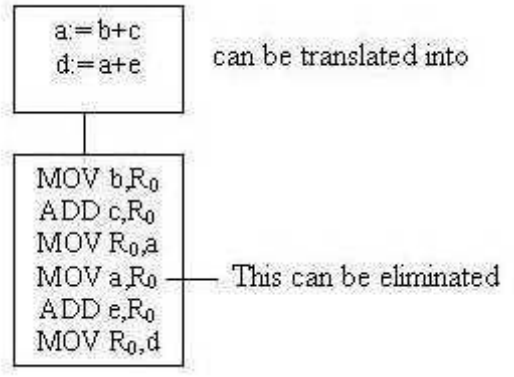
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

- Labels in three-address statements have to be converted to addresses of instructions. For example,
  j :**goto***i*generates jump instruction as follows :
  - if*i*<*j*, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
  - if*i*>*j*, the jump is forward. We must store on a list for quadruple*i* the location of the first machine instruction generated for quadruple*j*. When *i*is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.

- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :

Register allocation – the set of variables that will reside in registers at a point inthe program is selected.

- ***Register assignment*** – the specific register that a variable will reside in ispicked.

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair y – divisor
even register holds the remainder odd
register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three -address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

The algorithmic sequence of *getreg* function can be,

1. if x value is in register that register is returned.
2. If (1) fails, new register is returned.
3. If (2) fails, and the operation needs a special register, that register value is temporarily moved to the memory and the register is returned.
4. If (3) fails, finally memory location is returned.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

```
t : = a – b
u : = a – c
v : = t + u
d : = v + u
```

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : =t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements **a : = b [i]** and **a [i] : = b**

Statements	Code Generated
a : = b[i]	MOV b(Ri), R
a[i] : = b	MOV b, a(Ri)

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments **a : = *p** and ***p : = a**

Statements	**Code Generated**
a : = *p	MOV *R_p, a
*p : = a	MOV a, *R_p

Generating Code for Conditional Statements

Statement	**Code**
if x < y goto z	CMP x, y CJ<z /* jump to z if condition code is negative */

x : = y +z if x <0 goto z	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z


SUBBIAH EDUCATION TRUST

Runtime Environments:

A runtime environment, often referred to as a runtime system or runtime library, is a crucial component of a programming language's execution model. It provides the necessary infrastructure to run programs written in that language. The runtime environment includes various components and services that facilitate the execution of code. Here are the key components and functions of a runtime environment:

1. **Memory Management:**
   - The runtime environment manages memory allocation and deallocation for variables, objects, and data structures.
   - It handles memory leaks, freeing up memory that is no longer in use.

2. **Execution Stack:**
   - The runtime environment maintains an execution stack, often referred to as the call stack, which keeps track of function calls and their local variables.
   - It handles the allocation and deallocation of stack frames for function calls.

3. **Garbage Collection:**
   - In languages with automatic memory management, the runtime environment includes a garbage collector responsible for identifying and reclaiming memory that is no longer reachable.

4. **Type System:**
   - The runtime environment enforces the type system of the language, ensuring that operations are performed on compatible data types.
   - It may perform type checking and type coercion during runtime.

5. **Exception Handling:**
   - The runtime environment provides mechanisms for handling exceptions, including try-catch blocks or exception tables.
   - It ensures that exceptions are caught, and appropriate actions are taken, such as jumping to an exception handler.

6. **Dynamic Dispatch (Polymorphism):**
   - In object-oriented languages, the runtime environment supports dynamic method dispatch, allowing method calls on objects to be resolved at runtime based on their actual types.

7. **I/O Operations:**
   - The runtime environment provides functions or libraries for input and output operations, including reading from and writing to files, console, and network.

8. **Concurrency and Multithreading:**
   - In languages that support concurrency and multithreading, the runtime environment manages threads, synchronization, and thread-specific data.

9. **Dynamic Linking and Loading:**
   - The runtime environment may handle dynamic linking and loading of libraries or modules at runtime.
   - It resolves external references and links them to the running program.

10. **Environment Variables and Configuration:**
    - The runtime environment allows access to environment variables and system configuration settings, which can be useful for program configuration and customization.

11. **Standard Library:**
    - Most runtime environments include a standard library that provides common utility functions, data structures, and algorithms.
    - Programmers can leverage these libraries to perform various tasks without reinventing the wheel.

12. **Profiling and Debugging Tools:**
    - Some runtime environments come with profiling and debugging tools that help developers identify performance bottlenecks, memory issues, and logical errors.

13. **Security Features:**
    - The runtime environment may include security mechanisms to protect against common vulnerabilities like buffer overflows, unauthorized access, and code injection attacks.

14. **Platform Abstraction:**
    - The runtime environment abstracts the underlying hardware and operating system, providing a consistent interface for program execution across different platforms.

15. **Resource Management:**
    - It manages system resources such as file handles, network sockets, and threads, ensuring they are properly allocated and released.

The specifics of a runtime environment can vary significantly depending on the programming language and the execution model. High-level languages like Java, Python, and C# have robust runtime environments that provide many of these features, while lower-level languages like C and C++ may have more minimal runtime environments with fewer abstractions. Understanding the runtime environment is crucial for both application developers and system programmers, as it impacts program behavior, performance, and resource usage.

Source Language Issues:

Source language issues refer to various considerations, challenges, and design decisions that arise when developing or working with a programming language. These issues encompass language design, syntax, semantics, and pragmatics. Here are some common source language issues:

1. **Syntax Design:**

- Syntax design involves defining the rules for writing valid programs in the language.
- Decisions include the choice of symbols, keywords, punctuation, and the structure of statements and expressions.
- Striking a balance between readability, expressiveness, and simplicity is critical.

2. **Semantics:**
- Semantics define the meaning of program constructs in a language.
- Designers need to consider how different language features behave, such as variable scoping, data types, and operations.
- Ensuring consistent and predictable behavior is essential.

3. **Type System:**
- The type system determines how data types are defined, manipulated, and used in the language.
- Issues include type safety, type compatibility, type inference, and the presence of primitive and user-defined types.

4. **Control Flow:**
- Designers must decide on the control flow constructs available in the language, including conditionals (if-else), loops (for, while), and branching (goto or labeled break/continue).
- Consideration of structured programming principles and readability is important.

5. **Concurrency and Parallelism:**
- Languages need to address the challenges of concurrent and parallel programming.
- Issues include support for threads, synchronization mechanisms, and avoiding race conditions.

6. **Error Handling:**
- The language should provide mechanisms for error handling, such as exceptions, error codes, or runtime checks.
- Designing a robust error-handling system can greatly improve program reliability.

7. **Memory Management:**
- Memory management issues include manual memory allocation and deallocation, garbage collection, and ownership models (e.g., reference counting).
- Decisions impact both performance and safety.

8. **Standard Library:**
- Designing a comprehensive standard library with well-documented functions and modules is essential.
- The library should cover essential tasks like file I/O, networking, data structures, and algorithms.

9. **Interoperability:**
- Languages often need to interact with other languages or libraries. Designers must consider how to facilitate inter-language communication.
- Issues include foreign function interfaces (FFIs), calling conventions, and data marshaling.

10. **Extensibility and Modularity:**
    - Languages should support modular programming through mechanisms like modules, namespaces, and packages.
    - Extensibility via libraries or plugins is also important for language ecosystems.

11. **Backward Compatibility:**
    - Maintaining backward compatibility is crucial when evolving a language.
    - Decisions about how to handle deprecated features and versioning impact the user base.

12. **Tooling and Development Environment:**
    - Designing a language includes considering the availability of development tools, such as compilers, debuggers, and IDE support.
    - A robust development ecosystem can greatly aid programmers.

13. **Documentation and Learning Curve:**
    - The language should have clear and accessible documentation for users, including tutorials, reference manuals, and examples.
    - Minimizing the learning curve for new programmers is essential for language adoption.

14. **Performance Optimization:**
    - Language designers must consider performance optimization techniques, including compiler optimizations, runtime profiling, and the ability to write high-performance code.

15. **Community and Ecosystem:**
    - Building and nurturing a community around the language is vital for its long-term success.
    - Supporting package managers, code repositories, and forums fosters collaboration and growth.

16. **Platform and Architecture Support:**
    - Designers must decide which platforms and architectures the language will support.
    - Cross-platform compatibility is often a desirable feature.

Source language issues vary depending on the goals and target audience of the language. Language designers need to carefully weigh these considerations to create a language that is expressive, efficient, and user-friendly. Additionally, language evolution and community feedback play significant roles in addressing and refining these issues over time.

Dynamic Storage Allocation:

Dynamic storage allocation refers to the process of allocating and managing memory at runtime, as opposed to static storage allocation, where memory is allocated at compile time. Dynamic allocation allows programs to work with data structures whose sizes or lifetimes are not known until the program is running. It's a critical aspect of memory management in

modern programming languages. Here are some key concepts and methods related to dynamic storage allocation:

1. **Heap Memory:
- Dynamic storage is typically allocated from a region of memory known as the heap.
- The heap is a region of memory separate from the program's stack, and it is used for dynamic allocation because it can grow and shrink as needed during program execution.

2. **Allocation and Deallocation:
- Allocation refers to the process of reserving a portion of memory from the heap to store data dynamically.
- Deallocation is the process of releasing memory that is no longer needed to be used by the program. This helps prevent memory leaks.

3. **Dynamic Memory Allocation Functions:
- Most programming languages provide functions or mechanisms to allocate memory dynamically, such as `malloc`, `calloc`, `realloc`, and `free` in C and C++.
- Languages like Python, Java, and C# have built-in memory management systems that automatically handle dynamic memory allocation and deallocation.

4. **Memory Leaks:
- Memory leaks occur when memory is allocated dynamically, but it is not properly deallocated when it is no longer needed.
- Over time, memory leaks can lead to the exhaustion of available memory, causing a program or system to become unstable or crash.

5. **Dangling Pointers:
- Dangling pointers are pointers that reference memory locations that have already been deallocated.
- Accessing a dangling pointer can lead to unpredictable behavior, including crashes or data corruption.

6. **Fragmentation:
- Fragmentation occurs when free memory is divided into small, non-contiguous blocks, making it challenging to allocate large contiguous blocks even when sufficient free memory is available.
- This can lead to inefficient memory utilization.

7. **Garbage Collection:
- Some languages, like Java, JavaScript, and C#, use automatic garbage collection to manage dynamic memory.
- Garbage collectors automatically identify and reclaim memory that is no longer reachable or referenced by the program.

8. **Reference Counting:
- Reference counting is a technique where each dynamically allocated object keeps track of the number of references to it.

- When the reference count reaches zero, meaning there are no more references to the object, the memory is automatically deallocated.

9. **Memory Pools and Custom Allocators:
- In some scenarios, custom memory allocation strategies, like memory pools or custom allocators, are employed to optimize memory allocation for specific use cases.

10. **Thread-Safety:
- In multi-threaded programs, dynamic memory allocation and deallocation should be thread-safe to avoid race conditions and data corruption.
- Thread-safe memory management often requires synchronization mechanisms.

11. **Memory Overhead:
- Dynamic memory allocation may come with memory overhead due to bookkeeping information that tracks allocated blocks.
- Efficient management of memory overhead is important for minimizing wasted memory.

Dynamic storage allocation is a powerful feature in programming that enables flexible data structures and efficient memory utilisation. However, it also requires careful attention to memory management practices to avoid common pitfalls like memory leaks and dangling pointers. Different programming languages and environments offer various tools and techniques for managing dynamic memory effectively.

Optimal Code Generation for Expressions:

Optimal code generation for expressions is a critical aspect of compiler design and code optimization. The goal is to generate machine code or intermediate code that executes expressions efficiently while minimizing runtime computation and memory usage. Here are some techniques and strategies for achieving optimal code generation for expressions:

1. Use Efficient Data Structures:
- Choose appropriate data structures to represent expressions during compilation, such as abstract syntax trees (ASTs) or intermediate representations (IRs).
- These data structures should facilitate easy traversal and manipulation of the expression tree.

2. Constant Folding:
- Identify constant subexpressions within an expression and compute their values at compile time.
- Replace the constant subexpressions with their computed values to reduce runtime computation.

3. Algebraic Simplification:
- Apply algebraic simplification rules to reduce the complexity of expressions.
- For example, simplify `x + 0` to `x`, `x * 1` to `x`, or `x - x` to `0`.

4. Strength Reduction:

- Replace expensive operations with less expensive equivalents.
- For example, replace multiplication with shifts or divisions with multiplications by reciprocals.

5. Common Subexpression Elimination (CSE):
- Identify and eliminate redundant calculations of the same subexpression within an expression.
- Replace repeated calculations with a single computation and reuse the result.

6. Register Allocation:
- Allocate registers efficiently for intermediate values used in expressions.
- Minimize the need to store intermediate results in memory by keeping them in registers.

7. Instruction Selection:
- Choose the most efficient machine instructions for performing operations in an expression.
- Use SIMD (Single Instruction, Multiple Data) instructions where applicable for vectorized operations.

8. Optimize Conditional Branching:
- Optimize branching conditions in expressions to reduce the number of branches and improve branch prediction.
- Use techniques like branch folding to simplify conditional expressions.

9. Loop-Invariant Code Motion:
- Identify expressions within loops that do not change during loop iterations (loop-invariant expressions).
- Move these expressions outside the loop to avoid redundant computations.

10. Inline Function Calls:
- For small, frequently called functions or expressions, consider inlining them to eliminate the function call overhead.

11. Compiler Optimizations:
- Take advantage of compiler optimizations, such as loop optimization, constant propagation, and dead code elimination, to improve code quality for expressions.

12. Target-Specific Optimization:
- Consider target-specific optimizations that take advantage of the architecture's features, such as instruction pipelining or parallel execution units.

13. Profiling and Feedback-Directed Optimization:
- Use profiling data to identify hotspots in the code, especially in frequently executed expressions.
- Apply optimization techniques selectively to focus on areas of the code where they have the most impact.

14. Compiler Flags and Options:

- Utilize compiler flags and options to enable specific optimizations or control the optimization level for expressions.

15. Benchmarking and Testing:
- Benchmark the generated code to measure its performance against expected goals.
- Thoroughly test the optimized code to ensure correctness.

Optimizing code generation for expressions is a balance between code size, execution speed, and memory usage. It often requires trade-offs and considerations specific to the target architecture and compiler. Compiler writers use a combination of the above techniques to generate efficient code for expressions in a way that benefits the overall performance of programs.

Dynamic Programming Code Generation:

Dynamic programming is a technique used to solve optimization problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the results in a table or cache. This approach can be used in code generation to efficiently generate code for complex computations or transformations. Here's how dynamic programming can be applied to code generation:

1. Define the Problem:
- Clearly define the code generation problem you want to solve. This could be optimizing code generation for mathematical expressions, parsing, or any other task.

2. Identify Subproblems:
- Break down the main code generation problem into smaller, overlapping subproblems. Each subproblem should be a simpler version of the main problem.

3. Define Recurrence Relations:
- Determine how the solutions to subproblems relate to each other. This is typically done through recurrence relations or equations.
- Define the base cases, which are the simplest subproblems that can be solved directly.

4. Create a Table or Cache:
- Create a data structure, often a table or cache, to store the results of solved subproblems. This is used to avoid redundant computations.
- The dimensions of the table should correspond to the parameters or variables that define the subproblems.

5. Bottom-Up or Top-Down Approach:
- Dynamic programming can be implemented using a bottom-up approach, where you solve the smallest subproblems first and build up to the main problem, or a top-down approach, where you start with the main problem and recursively solve smaller subproblems.
- The choice depends on the problem and the structure of the recurrence relations.

6. Memoization (Top-Down Approach):
- In the top-down approach, use memoization to cache the results of subproblems as they are solved.
- Before solving a subproblem, check if its solution is already in the cache. If so, retrieve the cached result instead of recomputing it.

7. Table Filling (Bottom-Up Approach):
- In the bottom-up approach, iteratively fill in the table from the base cases to the main problem.
- Each cell in the table represents the solution to a subproblem.
- Use previously computed solutions to calculate solutions for larger subproblems.

8. Optimize and Store Code:
- As you solve subproblems and populate the table or cache, you can simultaneously generate and optimize code.
- The code generated for each subproblem should contribute to the final code for the main problem.

9. Retrieve Final Solution:
- Once the main problem's solution is computed and stored in the table or cache, retrieve it to obtain the final generated code.

10. Handle Specific Code Generation Logic:
- Depending on the code generation problem, you may need to implement specific logic for generating code snippets or performing transformations within the dynamic programming framework.

11. Analyze Complexity:
- Analyze the time and space complexity of your dynamic programming-based code generation approach. Ensure it meets performance requirements.

Dynamic programming-based code generation is particularly useful for problems involving complex recursive computations, optimization tasks, or problems with overlapping subproblems. By breaking down the problem into manageable parts and efficiently storing and reusing results, dynamic programming can lead to more efficient and optimized code generation processes.

UNIT V CODE OPTIMIZATION **8**

Principal Sources of Optimization – Peep-hole optimization - DAG- Optimization of Basic Blocks- Global Data Flow Analysis - Efficient Data Flow Algorithm.

CODE OPTIMIZATION

INTRODUCTION

The code produced by the straight forward 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. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

Machine independent optimizations:

Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speed up programs by a measurable amount.
- The transformation must be worth the effort

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

 consider the following source code for dot product of two vectors a and b of length 20

```
begin
        prod :=0;
        i:=1; do
        begin
                prod :=prod+ a[i]* b[i]; i
                :=i+1;
        end
        while i <= 20
end
```

- The three -address code for the above source program is given as :

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4* i
(4)   t2 := a[t1]      /*compute a[i] */
(5)   t3 := 4*i
(6)   t4 := b[t3]      /*compute b[i] */
(7)   t5 := t2*t4
(8)   t6 := prod+t5
(9)   prod := t6
(10)  t7 := i+1
(11)  i := t7
      if i<=20 goto
(12)  (3)
```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

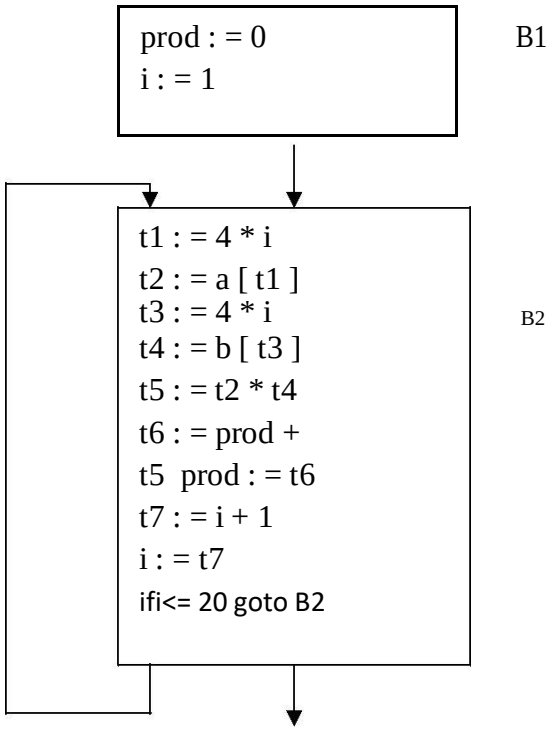
Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:

B1 is the *initial* node. B2 immediately follows B1, so there is an edge from B1 to B2

- The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).

B1 is the *predecessor* of B2, and B2 is a *successor* of B1



THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
    1. Leaves are labeled by unique identifiers, either variable names or constants.
    2. Interior nodes are labeled by an operator symbol.
    3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.

- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i)x := y OP z

Case (ii)x := OP y

Case (iii)x := y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z) . (Checkingfor common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

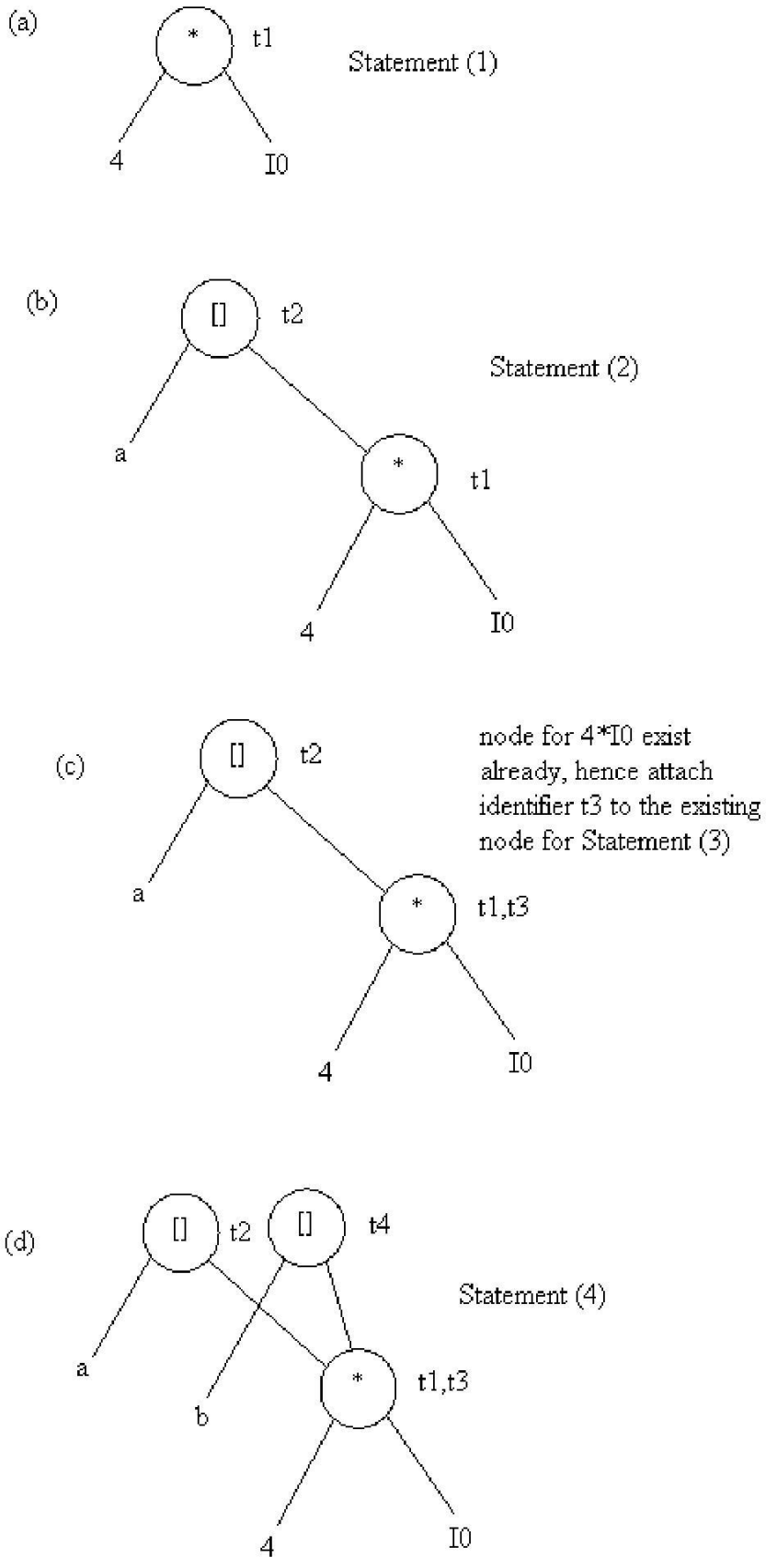
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

identifiers for the noden found in step 2 and set node(x) to n.

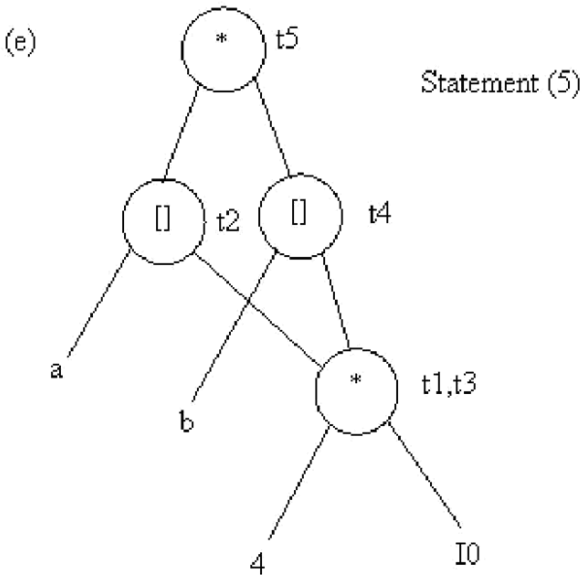
Example: Consider the block of three- address statements:

1. $t_1 := 4* i$
2. $t_2 := a[t_1]$
3. $t_3 := 4* i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2*t_4$
6. $t_6 := prod+t_5$
7. $prod := t_6$
8. $t_7 := i+1$
9. $i := t_7$
10. if i<=20 goto (1)

Stages in DAG Construction

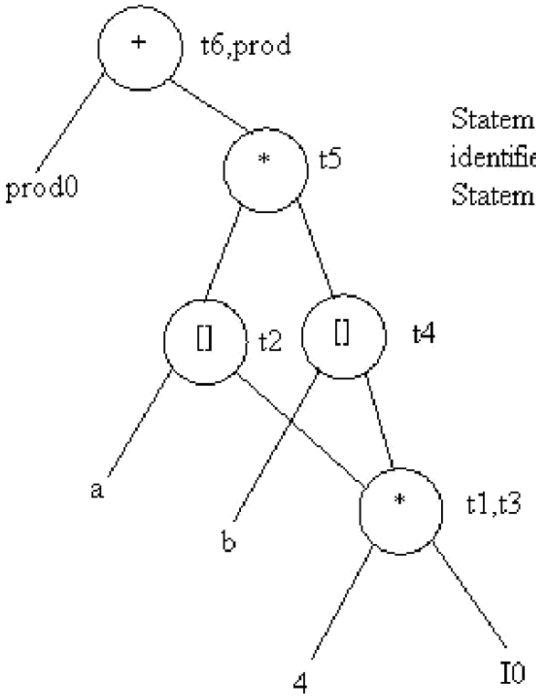


(e)



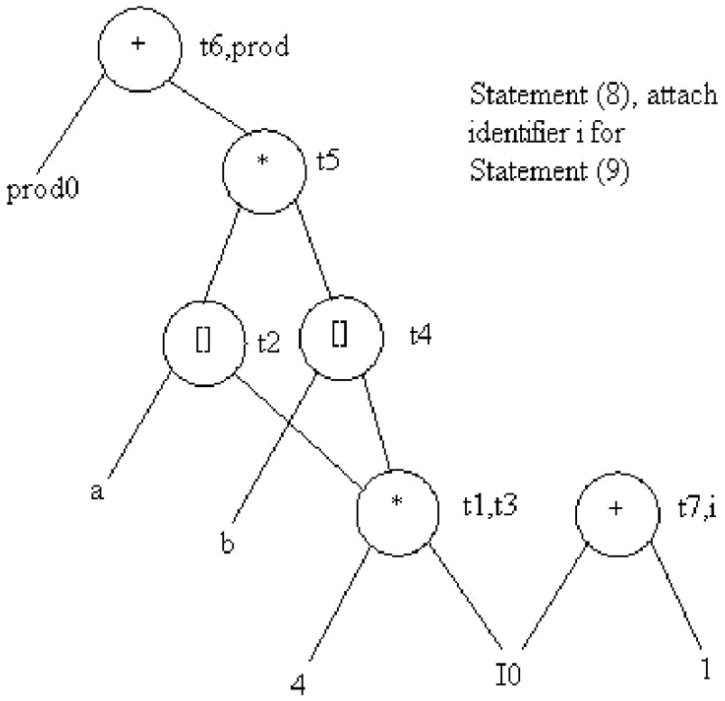
Statement (5)

(f)



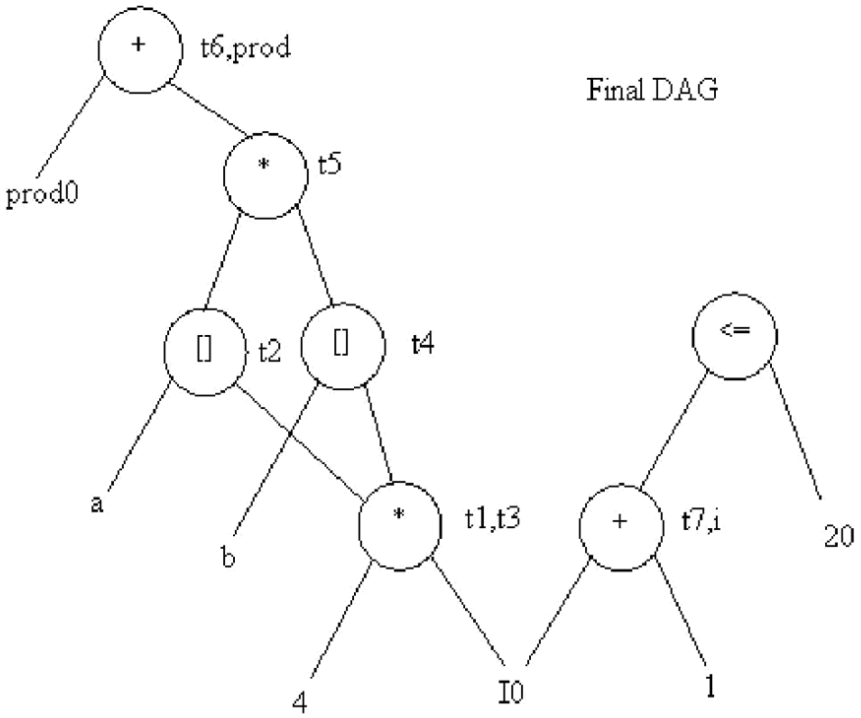
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations

  - ✓ Common sub expression elimination,
  - ✓ Copy propagation,
  - ✓ Dead-code elimination, and
  - ✓ Constant folding

  are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➢ Common Sub expressions elimination:

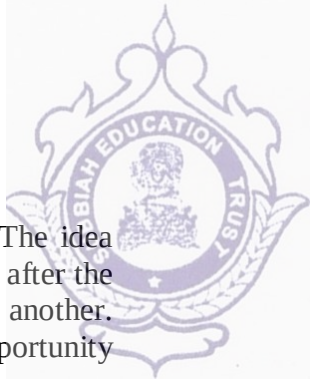
- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
- For example

t_1: = 4*i
t_2: = a
[t1] t_3: =
4*j t_4: =
4*i t_5: =
n
t_6: = b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: =
4*it_2: =
a [t_1]t_3:
= 4*jt_5:
= n
t_6: = b [t_1] +t_5

The common sub expression t 4: =4*i is eliminated as its computation is already in t_1.



And value of i is not been changed from definition to use.

- **Copy Propagation:**

– Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

– For

example:

```
x=Pi;

……
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

- **Dead-Code Eliminations:**

- 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. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
    i=0;
if(i=1)
{
 a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

- **Constant folding**:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

- One advantage of copy propagation is that it often turns the copy statement into dead code.

✓ For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. 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 that loop.
- Three techniques are important for loop optimization:

- code motion, which moves code outside a loop;
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

➢ **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 (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
Code motion will result in the equivalent of
 t= limit-2;
while (i<=t)   /* statement does not change limit or t */
```

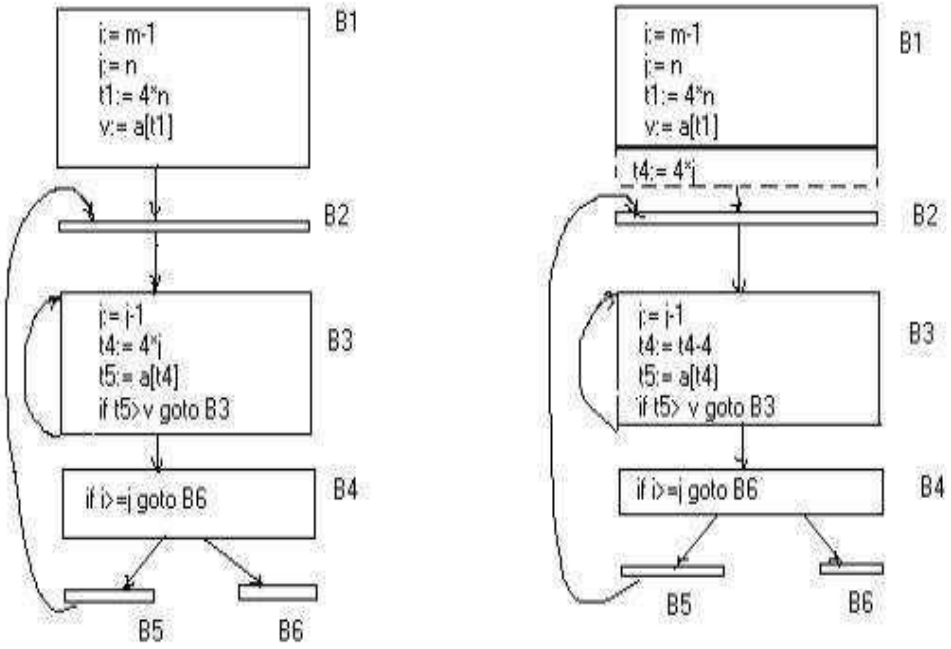
➢ **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

As the relationship $t_4:=4*j$ surely holds after such an assignment to $t_4$in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j -1 the relationship $t_4:= 4*j-4$ must hold. We may therefore replace the assignment $t_4:= 4*j$ by $t_4:= t_4-4$. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t_4=4*j$ on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.



Before **after**

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➢ **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short

sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

We shall give the following examples of program transformations that are characteristic of peephole optimizations:

- ✓ Redundant-instructions elimination
- ✓ Flow-of-control optimizations
- ✓ Algebraic simplifications
- ✓ Use of machine idioms
- ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a**is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0
….
If ( debug ) {
Print debugging information     }
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L1

 goto L2

L1: print debugging information

L2:.....(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
        If debug≠1 goto L2

        Print debugging information

    L2:.....(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

```
        If debug≠0 goto L2

        Print debugging information

    L2:.....(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
        goto L1

        ….

    L1: goto L2
```

by the sequence

```
        goto L2

        ….

    L1: goto L2
```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
    if a < b goto L1
      ….
    L1: goto L2
```

can be replaced by

```
    If a < b goto L2
```

….

L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

……..

L1: if a < b goto L2

L3:..... (1)

- May be replaced by

If a < b goto L2

goto L3

…….

L3:..... (2)

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0 Or

x := x * 1

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation

routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like
- i:=i+1 → i++
- i:=i-1 → i- -

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

➢ **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a=b+c
b=a-d
c=b+c
d=a-d
```

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error -correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

- **Renaming of temporary variables:**

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.

In this we can transform a basic block to its equivalent block called normal-form block

Interchange of two independent adjacent statements:

- Two statements
  t_1:=b+c
  t_2:=x+y
  can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

  a :=b+c
  e :=c+d+b

  the following intermediate code may be generated:

a :=b+c
t :=c+d
e :=t+b

- Example:

  x:=x+0 can be removed

  x:=y**2 can be replaced by a cheaper statement x:=y*y

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

- The details of how data-flow equations are set and solved depend on three factors.

✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as

the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each i between 1 and n-1, either

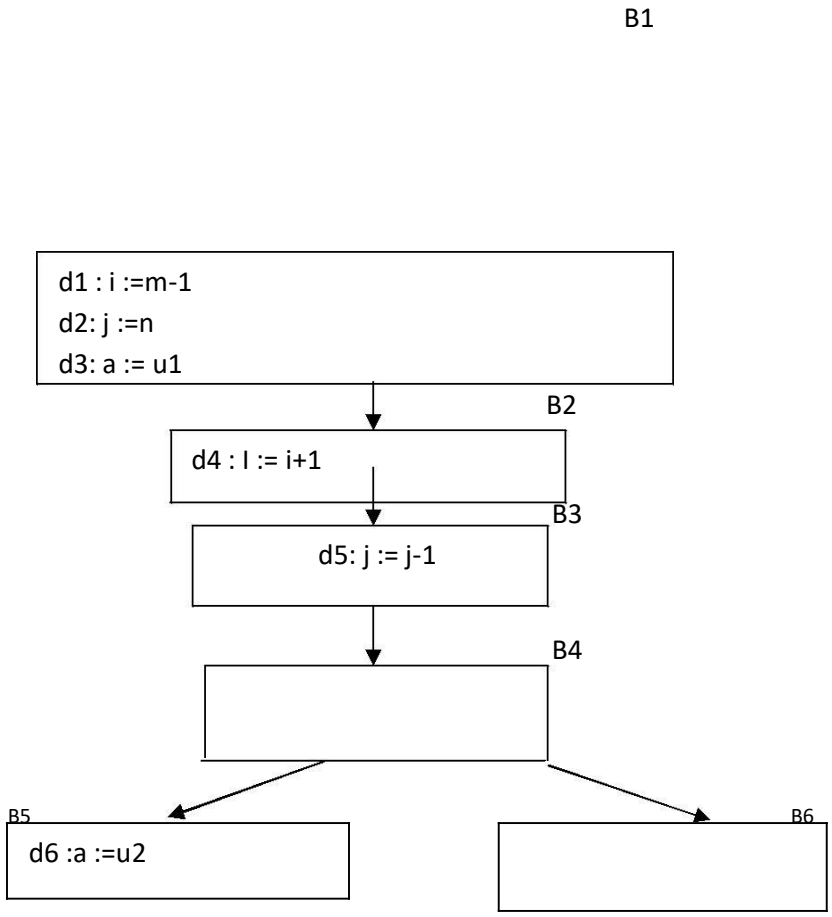
✓ P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

3. A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

4. These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:



- ✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
- ✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

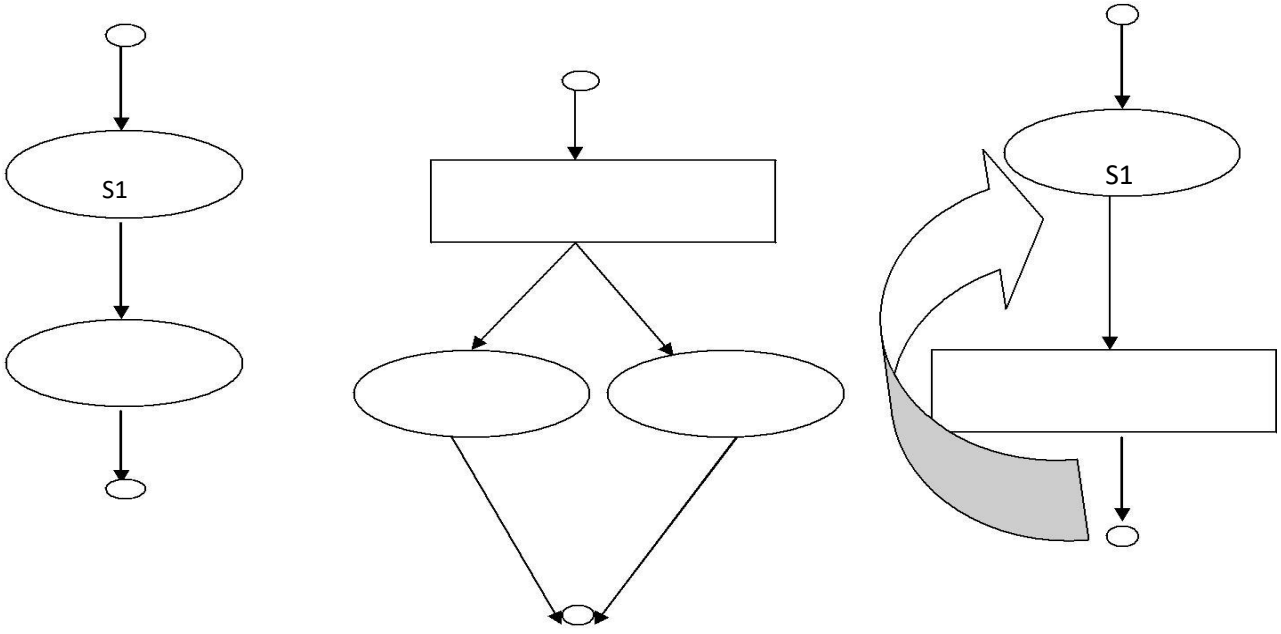
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  S ⟶ id: = E| S; S | if E then S else S | do S while E

  E⟶id + id| id

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



If E goto s1

S2

S1 S2

If E goto s1

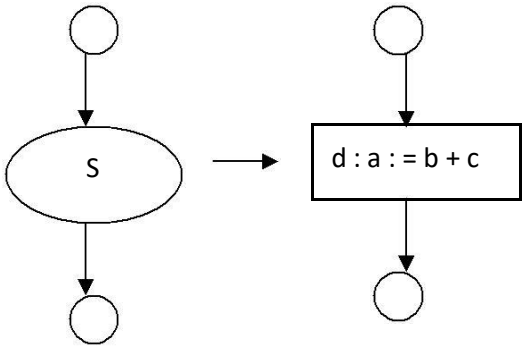
S1 ; S2

IF E then S1 else S2 do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

a)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

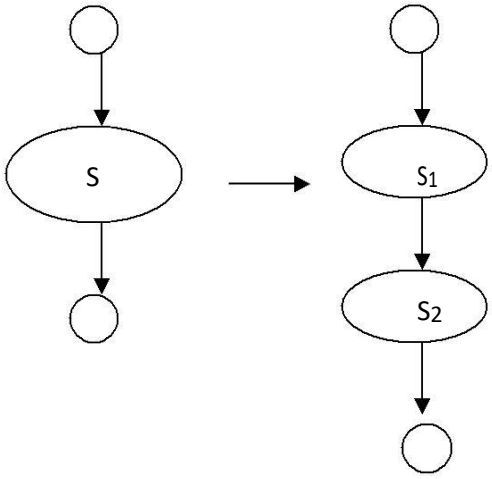
- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

gen[S]={d}

Kill[S] = D_a – {d}, d "kills" all other definitions of a, so we write

Where, D_a is the set of all definitions in the program for variable a.

b)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])

Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

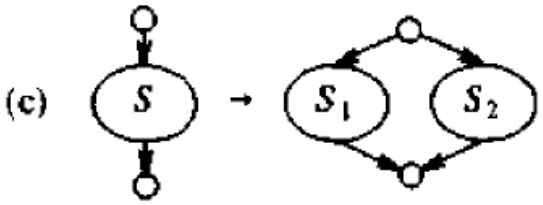
in [S_1] = in [S] in
[S_2] = out [S_1] out
[S] = out [S_2]

Under what circumstances is definition d generated by S=S_1; S_2? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write

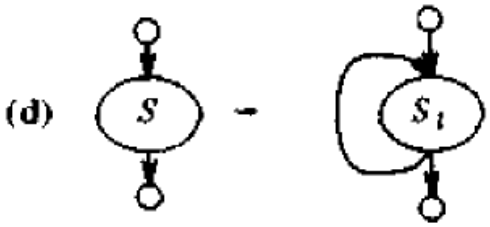
gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])

- Similar reasoning applies to the killing of a definition, so we have

Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])



$$gen[S] = gen[S_1] \cup gen[S_2]$$
$$kill[S] = kill[S_1] \cap kill[S_2]$$
$$in[S_1] = in[S]$$
$$in[S_2] = in[S]$$
$$out[S] = out[S_1] \cup out[S_2]$$



$$gen[S] = gen[S_1]$$
$$kill[S] = kill[S_1]$$
$$in[S_1] = in[S] \cup gen[S_1]'$$
$$out[S] = out[S_1]$$

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

0. Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
1. However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.
2. The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S

without following paths outside S.

3. Assuming we know in[S] we compute out by equation, that is

   Out[S] = gen[S] U (in[S] - kill[S])

- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

  In[S_1] = in[S_2] = in[S]

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

2. The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.

3. Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.

- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods
- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

Recent Trends in Compiler Design:

Compiler design is an ever-evolving field with continuous advancements and trends driven by changes in programming languages, hardware architectures, and software development practices. As of my last knowledge update in September 2021, here are some recent trends and developments in compiler design:

1. **Support for New Programming Languages:**
   - Compiler designers are adapting to support new programming languages and language features. This includes domain-specific languages (DSLs), WebAssembly (Wasm), and emerging languages like Rust and Julia.

2. **Language Server Protocol (LSP):**
   - LSP is becoming increasingly popular for integrating compilers with development environments. It provides features like autocompletion, code navigation, and real-time error checking.

3. **Just-In-Time (JIT) Compilation:**
   - JIT compilation is used in dynamic languages like Python, JavaScript (V8 engine), and the Java Virtual Machine (JVM) to improve execution speed. Compiler designers are working on optimizing JIT compilation techniques.

4. **Parallelism and Concurrency:**
   - Modern hardware supports multi-core processors, and compilers are incorporating techniques for automatic parallelization and concurrency to make better use of these resources.

5. **Machine Learning and Compiler Optimization:**
   - Machine learning is being applied to compiler optimization. AI-driven techniques can help identify performance bottlenecks and generate more efficient code.

6. **Energy-Efficient Compilation:**
   - With the growing concern for energy efficiency, some compiler design efforts are focused on generating code that minimizes power consumption while maintaining performance.

7. **Quantum Computing:**
   - Quantum programming languages are emerging, and compilers for quantum computing are being developed to translate high-level quantum algorithms into machine-specific instructions.

8. **Security and Safety:**
   - Ensuring secure and safe code generation is critical. Some trends include the development of languages and compilers with built-in security features and the use of formal methods for verification.

9. **WebAssembly (Wasm):**
   - WebAssembly is gaining traction as a portable compilation target for web browsers. Compilers are being developed to generate Wasm code from various source languages.

10. **Compiler Optimizations for GPUs:**
    - As GPUs are increasingly used for general-purpose computing (GPGPU), compilers are being enhanced to generate code optimized for GPU architectures.

11. **Static Analysis Tools:**
    - Static analysis tools and techniques are integrated into compilers to catch bugs, security vulnerabilities, and performance issues during compilation.

12. **Hybrid Compilation Techniques:**
    - Some compilers use a hybrid approach, combining ahead-of-time (AOT) and just-in-time (JIT) compilation to balance startup performance and runtime optimization.

13. **Open-Source Compiler Projects:**
    - Many open-source compiler projects continue to thrive, providing a collaborative environment for compiler innovation. Projects like LLVM and GCC are actively developed and adopted.

14. **Low-Level Optimization:**
    - Compiler designers are exploring low-level optimizations, including vectorization, loop unrolling, and instruction scheduling, to improve the performance of generated code.

15. **Integration with IDEs:**
    - Integration between compilers and integrated development environments (IDEs) is becoming tighter, offering features like real-time code analysis, refactoring support, and intelligent code completion.

It's important to note that the field of compiler design continues to evolve, and new trends may have emerged since my last update. Staying current with these trends is essential for compiler designers, language developers, and anyone involved in software development, as they can significantly impact the efficiency, security, and maintainability of software systems.

CS3501-Compiler Design

Question Bank

Unit-I

INTRODUCTION TO COMPILERS & LEXICAL ANALYSIS

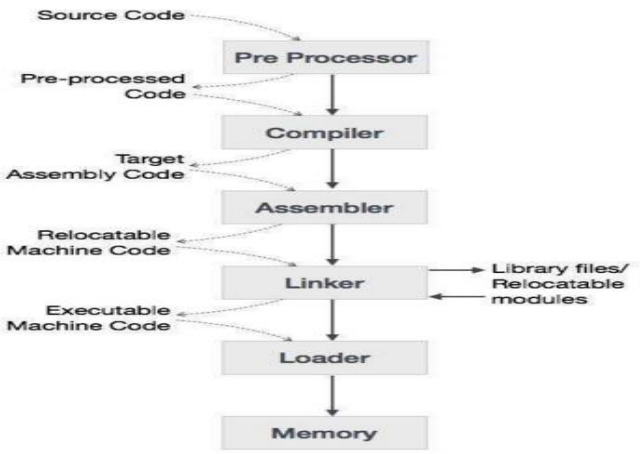
Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

Part-A

1. **Mention few cousins of the Compiler. M/J 2012**
   - Pre-processor
   - Assembler
   - Loader
   - Link-Editor

2. **What are the two parts of a compilation? Explain. M/J'2016**
   There are two parts to compilation: **analysis and synthesis**.
   i)Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
   ii) Synthesis part constructs the desired target program from the intermediate representation. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a **tree.** A special kind of tree called **a syntax tree**

3. **Illustrate diagrammatically how a language is processed. M/J'2016**



4. **What are the possible error recovery actions in lexical analyzer? M/J 2012, A/ M'2015**
   1. Deleting an extraneous character
   2. Inserting a missing character
   3. Replacing an incorrect character by a correct character
   4. Transposing two adjacent characters

5. Define tokens, Patterns and lexemes M/J'2013, N/D'2016

Tokens: A token is a pair consisting of a token name and an optional attribute value.A token name is an abstract symbol representing a kind of lexical unit eg:a particular keyword or a sequence of input character denoting an identifier.

Patterns: A Pattern is a rule describing the set of lexemes that can represent a particular token in source program. Ex: **relation : all six relational operator**

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for the token. For example in the Pascal's statement const pi = 3.1416; the substring pi is a lexeme for the token identifier

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except"

6. Mention the issues in a lexical analyzer. M/J'2013

1)Simpler design is the most important consideration.

- A parser including the conventions for comments and white space is significantly more complex

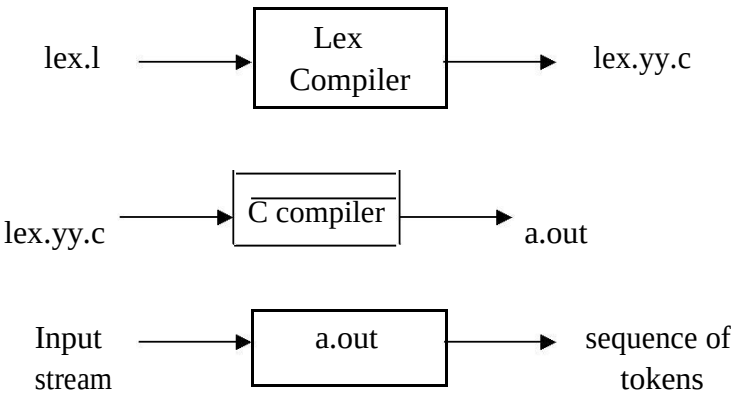
2) Compiler efficiency is improved.

- Specialized buffering techniques for reading input characters and processing tokens

3) Compiler portability is enhanced.

- Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

7. What are the components of LEX? N/D'2015



8. State any two reasons as to why phases of compiler should be grouped. M/J'2014

(i) Operation of compilation.-Analysis (Front End) and Synthesis(Back End)

(ii) Number of Passes- One Pass and Multi-Pass

9. Define Lexeme. M/J'2014

A lexeme is a sequence of characters in the source program that is matched by the pattern for the token. For example in the Pascal's statement
const pi = 3.1416; the substring **pi** is a lexeme for the token "identifier"

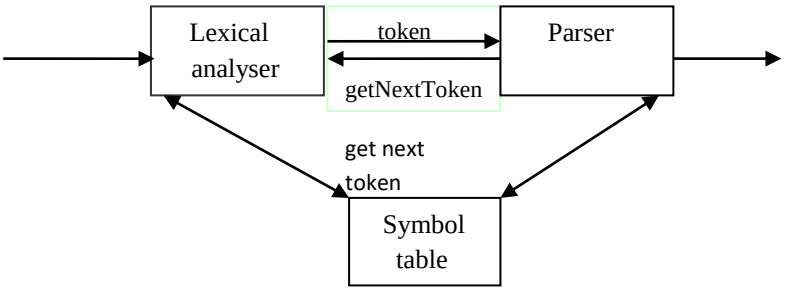
10. List the operation on languages. M/J'2016

- A **language** is any countable set of strings over some fixed alphabet.

Operations on languages:

- The following are the operations that can be applied to languages:
  1.Union
  2.Concatenation
  3.Kleene closure
  4.Positive closure

11. State the interactions between the lexical analyzer and the parser. N/D'2015



- Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis.
- As in the figure, upon receiving a **"get next token"** command from the parser the lexical analyzer reads input characters until it can identify the next token.

12. Define error recovery strategies. N/D'2015

- Panic mode Recovery
- Phrase level
- Error Production
- Global Correction
- Other Error recovery action
  1)Deleting an extraneous character.
  2) Inserting a missing character.
  3)Replacing an incorrect character by a correct character.
  4)Transforming two adjacent characters

13. What is a symbol table? N/D'2016, M/J'2014

- 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 quickly.

- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.
- Ex: Var posi, init, rate : real;

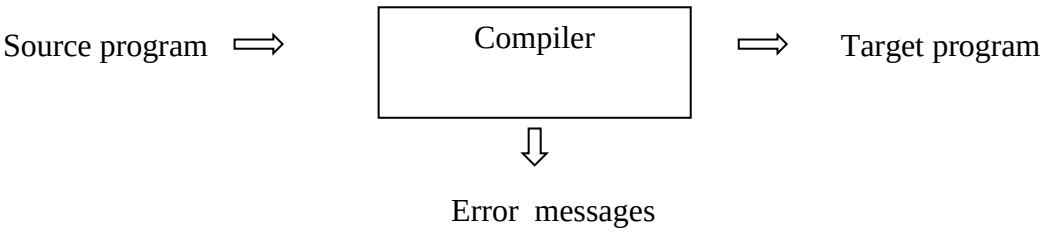
pos	Real	…
init	Real	…
rate	Real	…

14. List out various compiler construction tools. N/D'2016

- Scanner Generators
- Parser Generators
- Syntax-Directed Translation Engines
- Automatic Code Generators
- Data-Flow Engines

15. Define a compiler?

A Compiler is a program that reads a program written in one language (source language) and translate it into an equivalent program in another language (target language). While compilation, the reports to its user the presence of errors in the source program.



16. What are the functions of preprocessors?

- Macro processing
- File inclusion
- Rational preprocessors
- Language Extensions

17. Define Interpreter. APRIL/MAY 11

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig.

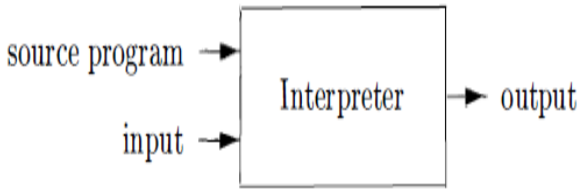


Figure 1.3: An interpreter

18. Describe Assembly code with an example.

Assembly code is a mnemonic version of machine code. In which names are used instead of binary codes for operations, and names are also given to memory addresses.

A typical sequence of assembly instructions might be

MOV a , R1 0001 01 00 00000000
ADD #2 , R1 0011 01 10 00000010
MOV R1 , b 0010 01 00 00000100

(i) In Which the first four bits are instruction are, with 0001, 0010,0011 standing for load, store and add.

(ii) The next two bits designate a register. (ie) 01-R1

(iii) The next two bits are 00-> ordinary address and 10-> immediate mode.

Last eight bits refer to memory address.

19. Compare compiler & interpreter.

S.No	Compiler	Interpreter
1	Program is analyzed only once & the code is generated.	The source program gets interpreted every time so that it can be executed.
2	Compilers produce object code.	It does not produce object code.
3	More efficient than interpreter.	Less efficient.
4	It is a complex program & requires high memory.	It is simpler & requires less memory.

20. What are the tools available for analysis part? Describe about any two.

- **STRUCTURE EDITORS:** The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- **Pretty Printers:** A pretty printer analyses a program and prints it in such a way that the structure of the program becomes clearly visible
- **Static Checkers:** A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- **Interpreters:** An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.

PART B

1. Explain the various phases of compiler in detail, with a neat sketch A/M2012, N/D 2012, M/J'2013, N/D'2013, M/J'2014,M/J'2016, N/D'2016

Contents:

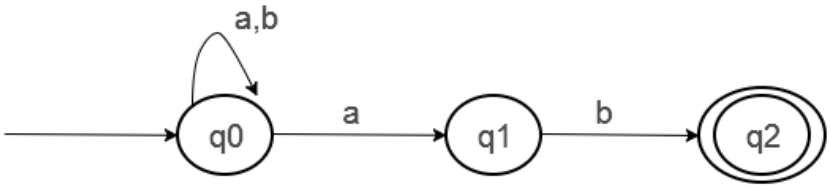
- Explain all the Phases of Compiler
- Neatly Represent the Diagram

2. **Explain language processing system with neat diagram.(8) (cousins of compiler) M/J'2016**

Contents:

- Explain all the Cousins of Compiler
- Neatly Represent the Diagram

3. **Convert the following NFA into Equivalent DFA**



Ans:

Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table

Step 2: Create the DFA's start state

Step 3: Create the DFA's transition table

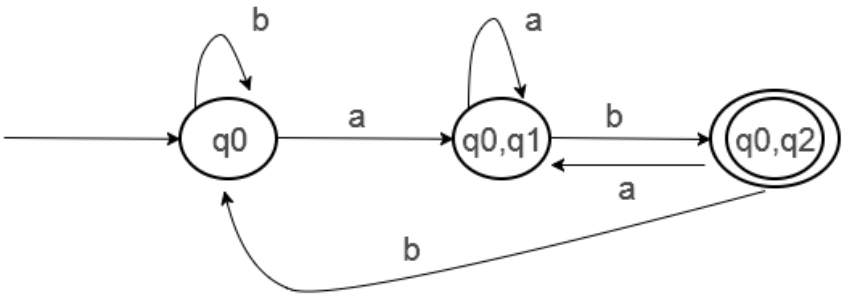
Step 4: Create the DFA's final states

Step 5: Simplify the DFA

Transition Table

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

Transition Diagram



4. **Convert the regular expression abb (a|b) to DFA using direct method and minimize it.(AU– April / May 2017)**

Contents:

- Define Regular expression
- Steps followed to the convention

- Follow the Standard Rules
- Convert the Regular Expression
- Write the Minimization Algorithm Steps and illustrate the converted Regular expression

5. **Draw the Transition Diagram for relational operators and unsigned numbers.(AU –April /May 2017)**
   Contents:
   - Define Finite Automata
   - Write down the Two Notations
   - Explain in detail the both notations
   - Give any one example

6. **Explain In detail About the Input Buffering Methods.**
   Contents:
   - Define Input Buffering
   - Write down the Two Methods of Buffering
   - Explain in detail the both methods with example
   - Write the Advantages and Disadvantages

7. **How to Specify the Tokens and explain the concepts. (8 Marks)**
   Contents:
   - Define Tokens
   - Write down the Three types of specifications
   - Explain in detail the both types with example

8. **Explain the Lex Tools in details (5 Marks)**
   Contents:
   - Define LEX
   - Explain in detail with example

Part–C (1 x 15 = 15 Marks)

1. What are the Phases of compiler? Explain the Phases in detail. Write down the output ofeach phases for the expression a: = b + c * 60(AU–April / May 2017)

Contents:
- Define Compiler
- Explain all the Phases of Compiler
- Neatly Represent the Diagram
- Write the Example Expression to be match with all phases ofcompiler

Unit-II

SYNTAX ANALYSIS

Role of Parser – Grammars – Context-free grammars – Writing a grammar Top Down Parsing – General Strategies – Recursive Descent Parser Predictive Parser-LL(1) – Parser-ShiftReduce Parser-LR Parser- LR (0)Item Construction of SLR Parsing Table – Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC tool – Design of a syntax Analyzer for a Sample Language

Part-A

1. Define Define Recursive Descent Parsing?

A Recursive Descent Parser (RDP) is a type of top-down parsing technique used incomputer science to analyze and process a language's syntax.

2. **Differentiate Top Down parsing and Bottom Up parsing?. A/M 2012**

Top Down parsing	Bottom Up parsing
It is a parsing strategy that first looks atthe highest level of the parse tree andworks down the parse tree by using therules of grammar.	It is a parsing strategy that first looks atthe lowest level of the parse tree andworks up the parse tree by using the rulesof grammar.
This parsing technique uses Left MostDerivation.	This parsing technique uses Right MostDerivation.
Example: Recursive Descent parser.	Example: Its Shift Reduce parser.

3. Compare Syntax tree and Parse tree. N/D 2012

Syntax tree:

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a **tree.** A special kind of tree called **a syntax tree** is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

Parse Tree:

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some nonterminal A and that the children of the node are labeled from left to right by symbols in the right side of the production by which this A was replaced in the derivation. The leaves of the parse tree are terminal symbols.

4. Write the rule to eliminate left recursion in a grammar. N/D 2012

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα |β it can be replaced with a sequence of two productions

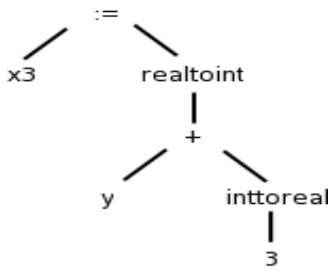
A → βA'

A'→ αA' | ε without changing the set of strings derivable from A

5. Mention the role of semantic analysis. N/D 2012

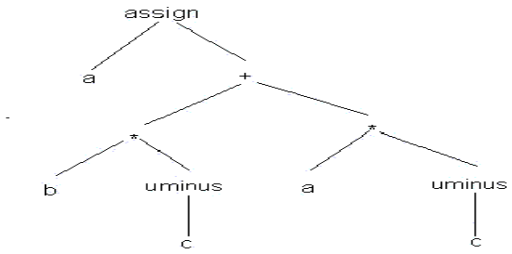
There is more to a front end than simply syntax. The compiler needs semantic information, e.g., the types (integer, real, pointer to array of integers, etc) of the objects involved. This enables checking for semantic errors and inserting type conversion where necessary.

For example, if y was declared to be a real and x3 an integer, We need to insert (unary, i.e., one operand) conversion operators "inttoreal" and "realtoint" as shown on the right.



6. Draw the syntax tree for the expression: a=b*-c +b*-c N/D 2012

A syntax tree depicts the natural hierarchical structure of a source program. A **DAG** (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree for the assignment statement a:=b*-c+b*-c appear in the figure.



.

Fig: Graphical Representation of a := b * -c + b * -c

7. Eliminate left recursion from the following grammar A->Ac/Aad/bd/€. M/J'2013

A->bdA'/A'

A'->cA'/adA'/€

8. Eliminate left recursion from the grammar S-> Aa | b ; A-> Ac | Sd | €. N/D'2013

Ans: S-> Aa | b ; A-> Ac|Aad |bd|€

Equivalent Rule:

S-> Aa | b

A->bdA'/A'

A'->cA'/adA'/€

9. Write a CF grammar to represent palindrome. N/D'2014

S->aSa | bSb | a | b| €

String={aba,bab,abba,ababa,...}

10. Write the role of parser. A/ M'2015

- The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It reports any syntax errors in the program.
- It also recovers from commonly occurring errors so that it can continue processing its input.

11. Write a grammar for branching statement. M/J'2016

S ->iEtS | iEtSeS | a

E -> b

12. Write the algorithm for FIRST and FOLLOW in parser. M/J'2016

First() : Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.

Follow () : Let A be a non-terminal. Follow(A) is the set of terminals a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).

Rules for follow( ):

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

13. Define LL (1) Grammar.

A grammar whose predictive parser has no multiply defined entries is known as LL(1)grammar. LL (1) means left to right scan of the input to generate the left most derivationby using 1 symbol of look ahead (can be extended to k symbol look ahead)

14. **Define parser.**

The parser is that phase of the compiler which takes a token string as input and with thehelp of existing grammar, converts it into the corresponding IntermediateRepresentation (IR). The parser is also known as Syntax Analyzer.

15. What is meant by handle pruning? N/D'2016

An Handle of a string is a sub string that matches the right side of production and whose reduction to the non terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

The process of obtaining rightmost derivation in reverse is known as Handle Pruning.

Consider the grammar: E → E+E/E*E/(E)/id

And the input string id1+id2*id3

The rightmost derivation is :

→ **E**+**E**

→ E+**E*E**

→ E+E***id3**

→ E+**id2***id 3

→ **id1**+id2***id 3

16. What are the errors identified in syntax analyzer?

- Syntax errors are errors in the program text; they may be either lexical or grammatical:

(a) A lexical error is a mistake in a lexeme, for examples, typing tehn instead of then, or missing off one of the quotes in a literal.

(b) A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing then).

(c) A syntactic error, such as an arithmetic expression with missing semicolon or unbalanced parenthesis.

17. Write the error recovery actions in parser?

- Panic mode recovery
- Phrase level recovery
- Error Production.
- Global Correction.

18. What do you mean by viable prefixes?

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. An equivalent definition of a viable prefix is that it is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

19. Write the limitations of Recursive decent parser.

- Back tracking is available
- Taking more time and more space
- Not efficient
- Reducing the performance of compiler.

20. List out the conflicts occurred in shift-reduce parsing.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

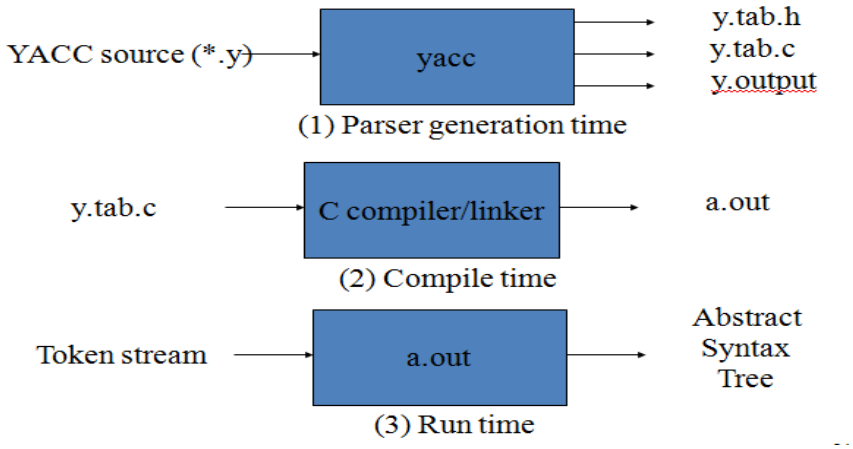
a. **Shift-reduce conflict**: The parser cannot decide whether to shift or to reduce.

b. **Reduce-reduce conflict**: The parser cannot decide which of several reductions to make.

21. Define YACC.

Yacc - Yet Another Compiler-Compiler

- Tool which will produce a parser for a given grammar.
- YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.



22. What is LL (1) grammar? Give the properties of LL (1) grammar.

L : Scan input from Left to Right
L : Construct a Leftmost Derivation
1 : Use "1" input symbol as lookahead in conjunction with stack to decide on the parsing action
LL(1) grammars == they have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

1. Grammar can't be ambiguous or left recursive
2. Grammar is LL(1) $\Leftrightarrow$when $A \rightarrow \alpha \mid \beta$
   a. α & β do not derive strings starting with the same terminal a
   b. Either α or β can derive $\in$, but not both.
   c. If β derives $\in$, then α does not derive any string beginning with a terminal in FOLLOW(A)

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

23. What are the disadvantages of operator precedence parsing?

Advantages of operator precedence parsing:

- It is easy to implement.
- Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

- It is hard to handle tokens like the minus sign (-) which has two different precedence.
- Only a small class of grammar can be parsed using operator-precedence parser.

Part-B

1. **Discuss in detail about the role of parser.**

Contents:

- Explain the process of Parser work
- Types of Parser and explain
- Neatly Represent the Diagram

2. **Write an Algorithm and construct SLR Parsing Table for the following context freegrammar. Check whether the string id + id id * is a valid string (AU–April / May 2013)**

   E→E+T | T
   T→T*F | F
   F→F*| a| b

   Contents:

   - Define SLR
   - Steps involved for SLR Parser
   - Neatly Represent the given CFG Grammar
   - Construct the Parsing table
   - Check the Input String with stack implementation

3. **Explain LR Parsing algorithm with example (AU–Nov / Dec 2017)**

   Contents:

   - Define LR
   - Differentiate LL vs LR
   - Explain the Concepts
   - Advantages & Disadvantages

4. **Explain in detail YACC (AU–Nov / Dec 2016)**

   Contents:

   - Define YACC
   - Explain the Concepts of Input and output files
   - Advantages & Disadvantages
   - Example programs

5. **Give the LALR for the given grammar.S->AAA->Aa|b**

   Contents:

   - Define LALR
   - Neatly Represent the given CFG Grammar
   - Construct the Parsing table

Part – C

1. **Construct Stack implementation of shift reduce parsing for the grammarE->E+EE->E*EE->(E)E->id and the input string id1+id2*id3**

   Contents:

   - Explain the Concepts of Shift Reduce parser
   - Neatly Represent the given CFG Grammar
   - Check the Input String with stack implementation

Unit-III

SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION

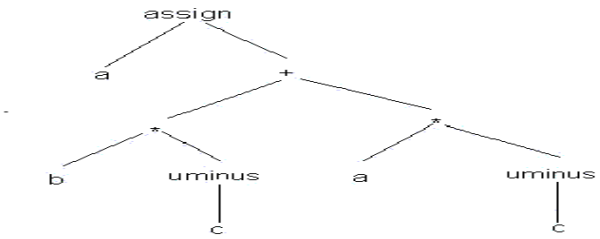
Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator – Type Systems-Specification of a simple type Checker Equivalence of Type Expressions-Type Conversions. Intermediate Languages: Syntax Tree, Three Address Code, Types and Declarations, Translation of Expressions, Type Checking, Back patching.

Part – A

1. **What are the various methods of implementing three address statements? M/J'2013**

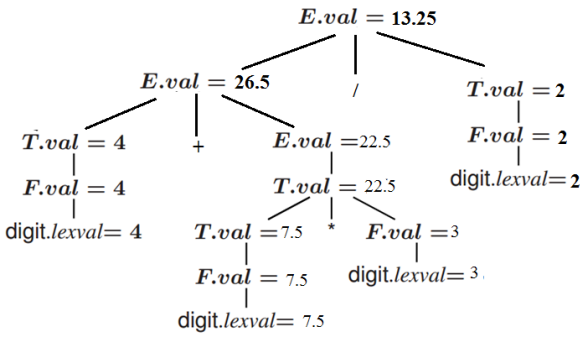
   There are three types of intermediate representation:-
   1. Syntax Trees
   2. Postfix notation
   3. Three Address Code

2. **Translate the arithmetic expression a * -(b+c) into syntax tree and postfix notation. N/D'201**



   Postfix Notation: a b c uminus * b c uminus * + assign

3. **Construct a decorated parse tree according to the syntax directed definition for the following input statement: (4+7.5*3)/2. A/ M'2015**



4. **Write the 3-address code x=*y ; a=&x, A/ M'2015**

   x:=*y MOV *R_y, x

   a:=&x MOV &R_x, a

5. **Write down syntax Directed definition of a simple desk calculator(AU –Nov / Dec 2016)**

Syntax Directed Definition (SDD) is a kind of abstract specification. The combination of context free grammar and semantic rules

6. **What are synthesized attributes? N/D'2015**
   a. Synthesized Attributes. They are computed from the values of the attributes of the children nodes.
   b. Inherited Attributes. They are computed from the values of the attributes of both the siblings and the parent nodes.

7. **Mention the rules for type checking (AU– April / May 2016)**

A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language.

8. **Write down syntax directed definition of a simple desk calculator. N/D'2016**

PRODUCTION	SEMANTIC RULES
L -> E n	print (E.val)
E -> E1 + T	E.val:=E1.val+T.val
E -> T	E.val:=T.val
T -> T1 * F	T.val:=T1.val*F.val
T -> F	T.val:=F.val
F -> (E)	F.val:=E.val
F -> digit	F.val=digit.lexval

Syntax-Directed Definition of a simple calculator

9. **Define syntax directed definition.**
   - Syntax Directed Definitions are a generalization of context-free grammars in which:
     1. Grammar symbols have an associated set of Attributes;
     2. Productions are associated with Semantic Rules for computing the values of attributes.
   - Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

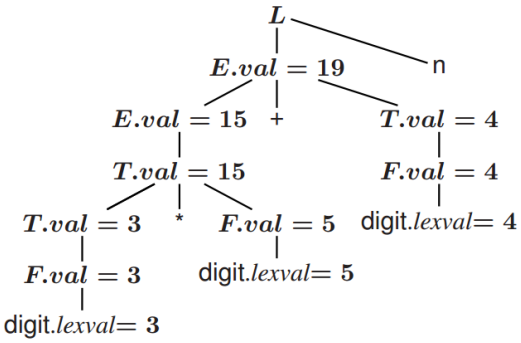
10. **Define S-Attributes.**

S-Attributed Definitions

Definition : An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.

• Evaluation Order. Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or Post Order, traversal of the parse-tree.

• Example. The above arithmetic grammar is an example of an S-Attribute d Definition. The annotated parse-tree for the input 3*5+4n is:



11. Define Dependency Graph.

- Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
  – There is a node for each attribute;
  – If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).
- Dependency Rule: If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

12. Mention the two rules of type checking. NOV/DEC2011

a. A type checker verifies that the type of a construct matches that expected by its context.
b. Type information gathered by a type checker may be needed when code is generated.

13. What is the significance of intermediate code? MAY/JUNE 14, APRIL/MAY 11, APRIL/MAY

- Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
- A machine-independent code optimizer can be applied to the intermediate representation.

14. What are the functions used to create the nodes of syntax trees?

- Mknode (op, left, right)
- Mkleaf (id,entry)
- Mkleaf (num, val)

15. What are the functions for constructing syntax trees for expressions?

- The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.
- Each node in a syntax tree can be implemented as a record with several fields.

16. Define back patching. (AU – Nov / Dec 2013)

Back patching is a technique for converting flow-of-control statements into a single pass. During the code generation process, it is the action of filling in blank labels with undetermined data. During bottom-up parsing, back patching is utilized to generate quadruples for boolean expressions.

17. Define type systems.

Type systems allow defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen statically (at compile time), dynamically (at run time), or as a combination of both.

18. Define a syntax-directed translation?

Syntax-directed translation specifies the translation of a construct in terms of Attributes associated with its syntactic components. Syntax-directed translation uses a context free grammar to specify the syntactic structure of the input. It is an input- output mapping.

19. Define an attribute. Give the types of an attribute?

An attribute may represent any quantity, with each grammar symbol, it associates a set of attributes and with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production. Example: a type, a value, a memory location etc.,

a. Synthesized attributes.

b. Inherited attributes.

20. Compare synthesized attributes and inherited attributes

Synthesized attributes	Inherited attributes
The production must have non-terminalas its head.	The production must have non-terminalas a symbol in its body
It can be evaluated during a singlebottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parsetree.
Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.

Part – B

1. **Explain in detail about Specification of a simple type checker. (AU – April / May 2017)**

Contents:

- Explain the concept of type checking expression
- Explain in Types checking statement
- Explain in Equivalence of type expression
- Neatly Represent the Diagram

2. **Discuss the following in detail about the Syntax Directed Definitions.Inherited Attributes and Synthesized attributes**

Contents:

- Define SDD
- Steps involved for SDD

- Explain in detail two methods of attributes
- Neatly Represent the given CFG Grammar

3. **Create variants of Syntax tree. Explain in detail about it with suitable examples**
   Contents:
   - Define Syntax Tree
   - Explain the Concepts
   - Explain in details the variants of Tree Structure

4. **State the rules for type checking with example**
   Contents:
   - Define Type checking
   - Explain the Concepts of two types of checking
   - Advantages & Disadvantages

5. **What is Type conversion? What are the two types of type conversion? Formulate therules for the type conversion.**
   Contents:
   - Explain the concept of type conversion or casting
   - Explain the Concepts of two types of conversion
   - Give an Example of each types.

Part– C

1. **Write the Translation scheme for translating assignment statement having scalarvariables and array reference to three - address statements(AU April / May 2013)**
   Contents:
   - Explain the Concepts of Three Address code
   - Write the applications
   - Implementation of Three address code and types
   - Give an Example of each types.

Unit-IV

RUN-TIME ENVIRONMENT AND CODE GENERATION

Runtime Environments – source language issues – Storage organization – Storage Allocation Strategies: Static, Stack and Heap allocation – Parameter Passing-Symbol Tables – Dynamic Storage Allocation – Issues in the Design of a code generator – Basic Blocks and Flow graphs – Design of a simple Code Generator –Optimal Code Generation for Expressions– Dynamic Programming Code Generation.

Part -A

1. Define Flow Graph. A/M 2012

Basic block

A sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch , are called basic blocks.

Flow graph

- The basic block and their successor relationships shown by a directed graph is called a flow graph.
- The nodes of a flow graph are the basic blocks.

2. How to perform register assignment for outer loop? A/M 2012

If an outer loop L1 contains an inner loop L2 , the names allocated registers in L2 need not be allocated registers in L1 - L2 . Similarly, if we choose to allocate x a register in L2 but not L1 , we must load x on entrance to L2 and store x on exit from L2 . We leave as an exercise the derivation of a criterion for selecting names to be allocated registers in an outer loop L, given that choices have already been made for all loops nested within L.

3. Give examples of static checks. M/J'2013

A compiler must check that the source program follows both the syntactic and semantic conventions of the source language. This checking, called static checking, ensures that certain kinds of programming errors will be detected and reported.

Examples of static checks include:

Type checking
Flow-of-control checks
Uniqueness checks
Name-related checks

4. List out the various storage allocation strategies. N/D'2014

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and de-allocates storage as needed at run time from a data area known as heap.

5. **What do you mean by binding of names? (AU April / May 2017)**

Name binding is the process of finding the declaration for each name that is explicitly or implicitly used in a template. The compiler may bind a name in the definition of a template, or it may bind a name at the instantiation of a template.

6. **What is the limitation of static allocation? APRIL/MAY 11**

1. The size of the data object and constraints on its position in memory must be known at compile time.
2. Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
3. Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

7. **List the different storage allocation strategies.**

The strategies are:

- Static allocation
- Stack allocation
- Heap allocation

8. **Brief about the techniques for parameter passing.(AU –Nov / Dec 2013)**

When using call by value, the compiler adds the R-value of the actual parameters that were passed to the calling procedure to the called procedure's activation record.

9. **What is dynamic scoping?**

In dynamic scoping a use of non-local variable refers to the non-local data declared in most recently called and still active procedure. Therefore each time new findings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

10. **What is heap allocation?**

Heap allocation is the most flexible allocation scheme. Allocation and deallocation ofmemory can be done at any time and any place depending upon the user's requirement.Heap allocation is used to allocate memory to the variables dynamically and when thevariables are no more used then claim it back

11. **How the activation record is pushed onto the stack.**

Activation record is used to manage the information needed by a single execution of aprocedure. An activation record is pushed into the stack when a procedure is called andit is popped when the control returns to the caller function.

12. **What are the properties of optimizing compiler?**

The source code should be such that it should produce minimum amount of target code.
There should not be any unreachable code.
Dead code should be completely removed from source language.
The optimizing compilers should apply following code improving transformations on source language.

a. common subexpression elimination

b. dead code elimination
c. code movement
d. strength reduction

13. What are the various ways to pass a parameter in a function?

- Call by value
- Call by reference
- Copy-restore
- Call by name

14. Suggest a suitable approach for computing hash function.

- Using hash function we should obtain exact locations of name in symbol table. The hash function should result in uniform distribution of names in symbol table.
- The hash function should be such that there will be minimum number of collisions. Collision is such a situation where hash function results in same location for storing the names.

15. Give short note about call-by-name?

Call by name, at every reference to a formal parameter in a procedure body the name of the corresponding actual parameter is evaluated. Access is then made to the effective parameter.

16. How parameters are passed to procedures in call-by-value method?

This mechanism transmits values of the parameters of call to the called program. The transfer is one way only and therefore the only way to returned can be the value of a function.

```
Main ( )
{ print (5);
} Int
Void print (int n)
{ printf (“%d”, n); }
```

17. Define static allocations and stack allocations

Static allocation is defined as lays out for all data objects at compile time.
Names are bound to storage as a program is compiled, so there is no need for a run time support package.
Stack allocation is defined as process in which manages the run time as a Stack. It is based on the idea of a control stack; storage is organized as a stack, and activation records are pushed and popped as activations begin and end.

18. Write the grammar for flow-of-control statements?

The following grammar generates the flow-of-control statements, if-then, if- then-else, and while-do statements.

```
S -> if E
then S1
| If E then S1 else S2
```

| While E do S1.

19. What are the control-flow constraints? N/D'2015

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
    goto L1
      ….
L1:  gotoL2
```

by the sequence

```
    goto  L2
      ….
L1: goto  L2
```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
    if a < b  goto L1
      ….
L1: goto L2
```

can be replaced by

```
    Ifa < b goto L2
    ….
    L1: goto L2
```

20. Write three address code sequence for the assignment statement. M/J'2016

D= (a-b)+(a-c)+(a-c)

temp1:= a-c

temp2:=a-b

temp3:=temp2+temp1

D:=temp3+temp1

Part-B

1. **Explain in detail the following with respect to code generation phase . (AU – Nov /Dec 2016)**

   Contents:

   - Explain the concept of type checking expression
   - Explain in Types checking statement
   - Explain in Equivalence of type expression
   - Neatly Represent the Diagram

2. **What are the different storage allocation strategies?(AU April / May 2017)**

   Contents:

   - Define Storage allocation
   - Write the three Strategies
   - Explain in detail all three strategies
   - Neatly Represent with advantage and disadvantage

3. **Explain in detail about the parameter passing (AU– April / May 2017)**
   Contents:
   - Define Syntax Tree
   - Explain the Concepts
   - Explain in details the variants of Tree Structure

4. **Explain about Runtime storage management.(AU –Nov / Dec 2017)**
   Contents:
   - Define Runtime Environment
   - Explain the Concepts of two types of checking
   - Advantages & Disadvantages

5. **Generate optimal code using Dynamic Programming techniques for the assignmentstatement , X: = (a/b - c) / d**. assume unit instruction costs **(AU Nov / Dec 2013)**
   Contents:
   - Explain the concept of type conversion or casting
   - Explain the Concepts of two types of conversion
   - Give an Example of each types.

Part– C

1. **Discuss the Various issues in Design of code generator (AU –April / May 2017)**
   Contents:
   - Explain the Concepts of Three Address code
   - Write the applications
   - Implementation of Three address code and types
   - Give an Example of each types.

Unit-V

CODE OPTIMIZATION

Principal Sources of Optimization – Peep-hole optimization – DAG- Optimization of Basic Blocks – Global Data Flow Analysis – Efficient Data Flow Algorithm – Recent trends in Compiler Design

Part -A

1. **List out two properties of reducible flow graph. A/M 2012**
   - Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated.
   - Data flow analysis problems can also be solved efficiently.

2. **List the advantage and application of DAG. N/D 2012, M/J'2013**
   - We can automatically detect common sub expressions.
   - We can determine the statements that compute the values, which could be used outside the block.
   - We can determine which identifiers have their values used in the block.

3. **What are the uses of register and address descriptor in code generator? N/D 2012**
   - A register descriptor keeps track of what is currently in each register.
   - An address descriptor keeps track of the location where the current value of the name can be found at run time.

4. **Define Live variable. N/D 2012**

   The framework is similar to reaching definitions, except that the transfer function runs backward. A variable is live at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.

5. **What is data flow analysis ? N/D 2012, N/D'2014**

   Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths. For example, one way to implement global common sub expression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program.

6. **Define Basic blocks and Flow graph. M/J'2013, N/D'2014**

   Basic block

   A sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch , are called basic blocks.

   Flow graph
   - The basic block and their successor relationships shown by a directed graph is called a flow graph.
   - The nodes of a flow graph are the basic blocks.

7. **What is constant folding ? M/J'2013**
   - We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known

as constant folding.

- One advantage of copy propagation is that it often turns the copy statement into dead code.

✓ For example,
a=3.14157/2 can be replaced by
a=1.570 thereby eliminating a division operation.

8. What are the properties of optimizing compiler? M/J'2013, N/D'2013, M/J'2016

1. Transformation must preserve the meaning of programs.
2. Transformation must, on the average, speed up the programs by a measurable amount
3. A Transformation must be worth the effort.

9. What is the Next-Use information? N/D'2013

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.
Input: Basic block B of three-address statements
Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.
Method: We start at the last statement of B and scan backwards.
1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

10. Define Loop unrolling with an example. N/D'2013

In region-based scheduling, the boundary of a loop iteration is a barrier to code motion. Operations from one iteration cannot overlap with those from another. One simple but highly effective technique to mitigate this problem is to unroll the loop a small number of times before code scheduling. A for-loop such as

```
for (i = 0; i < N; i++) {
SCi) ;
}
```

can be written as in Fig . 10.16(a) . Similarly, a repeat-loop such as

```
repeat
S;
unt il C
```

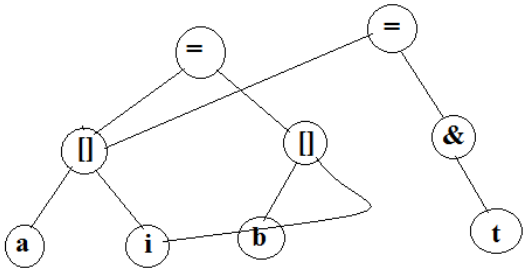
11. Name the techniques in loop optimization. M/J'2014

Three techniques are important for loop optimization:
code motion, which moves code outside a loop;
Induction-variable elimination, which we apply to replace variables from inner loop.
Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

12. How would you represent the dummy blocks with no statements indicated in global data flow analysis. M/J'2014

- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.

13. Draw DAG to represent a[i] =b[i]; a[i] = &t. N/D'2014



14. What role does the target machine play on the code generation phase of the compiler? A/ M'2015

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

The target computer is a byte-addressable machine with 4 bytes to a word.

It has *n* general-purpose registers, R0, R1, . . . , Rn-1.

It has two-address instructions of the form:

op source, destination

where, *op* is an op-code, and *source* and *destination* are data fields.

It has the following op-codes :

MOV (move *source* to *destination*)

ADD (add *source* to *destination*)

SUB (subtract *source* from *destination*)

The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

15. Generate code for the following C statement assuming three registers are available: x=a/(b+c)-d*(e+f) A/ M'2015

Three address code:

t1= e+f
t2=b+c
t3=a/t2
t4=d*t1
t5=t3-t4
x=t5

Assembly Code (Target Code):

MOV e, R1
ADD f,R1
MOV b,R2
ADD c,R2

```
MOV a,R3
DIV R3,R2
MUL d,R1
SUB R1,R2
MOV R2,x
```

16. Write the algorithm that orders the DAG nodes for generating optimal target code. A/ M'2015

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.
The algorithm shown below produces the ordering in reverse.

Algorithm:

1. **while** unlisted interior nodes remain **do begin**
2. select an unlisted node n, all of whose parents have been listed;
3. list n;
4. **while** the leftmost child m of n has no unlisted parents and is not a leaf **do begin**
5. list m;
6. n : = m

end
end

17. Define Dead code elimination. N/D'2015

A variable is live at a point in a program if its value can be used subsequently; otherwise that values never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

18. What are the issues in the design of code generator? N/D'2015

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

19. What are the global common sub expressions? N/D'2015

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

t_1: =4*i t_2: =a

[t1]

t_3: =4*j t_4:=4*i

t_5: =n

t_6: =b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: =4*i

t_2: =a [t_1] t_3:

=4*j

t_5: =n

t_6: =b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

20. What is DAG? M/J'2016

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  Leaves are labeled by unique identifiers, either variable names or constants.
  Interior nodes are labeled by an operator symbol.
  Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

21. What are the characteristics of peephole optimization.

- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- Characteristic of peephole optimizations:
  - Redundant-instructions elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms
  - Unreachable Code

Part -B

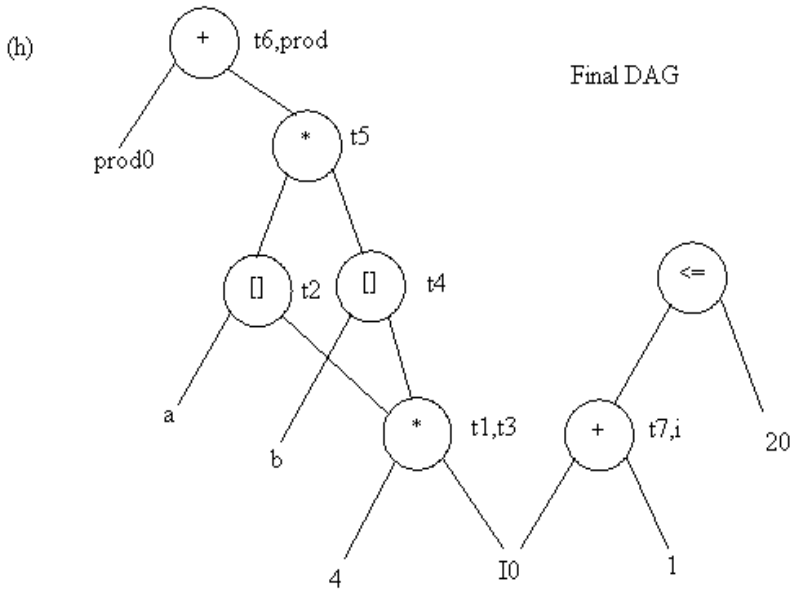
1. **Explain in detail about Global Data flow analysis of structural programs.(16) NOV/DEC2012, M/J'2013, N/D'2013, M/J'2014, N/D'2015, N/D'2016**

   Contents

   - Define Global Data flow analysis
   - Explain Points and Paths:
   - Draw Diagram Neatly
   - Write Data-flow analysis of structured programs:

2. **For the flow graph shown below, write the three address code and construct the DAG. (8) M/J'2013**

   (1) t1 = 4*i
   (2) t2 = a[t1]
   (3) t3 = 4 *i
   (4) t4 = b[t2]
   (5) t5= t2 * t4
   (6) t6 = prod + t5
   (7) prod = t6
   (8) t7 = i+1
   (9) i = t7
   (10) if i<=20 goto (1)



3. **Write short notes on structure preserving transformation of basic blocks.(8) NOV/DEC2013, N/D'2014, N/D'2015**

   Contents

   - Structure-Preserving Transformation
   - Dead-code elimination
   - Renaming temporary variables
   - Interchange of statements

4. **Construct DAG and three address statement for the following C Program. N/D'2013, N/D'2014**

```
i=1;
s=0;
while(i<=10)
{
s=s+a[i][i]
i=i+1;
}
```

   Contents

   - Define DAG
   - Write the three address code
   - Draw the DAG disgram

5. **Define a directed acyclic graph. Construct a DAG and write the sequence of instructions for the expression a+a*(b-c)+(b-c)*d. (16)MAY/JUNE 14**

   Contents

   - Construct a DAG and Equivalence of Instructions
   - Three address code for given ecpression
   - Write the Instruction code

6. **Explain the steps carried out for generating code from DAGs with an example.(8) N/D'2015**

   Contents

   - Rearranging the order
   - Write Generated code sequence for basic block
   - Write the algorithm
   - Give a suitable example

PART C

1. **Discuss about the following:**

   i). Copy Propagation ii) Dead-code Elimination and iii) Code motion(6) NOV/DEC2012, MAY/JUNE 2012

   Contents

   - Copy Propagation
   - Dead-Code Eliminations
   - Constant folding
   - Loop Optimizations
   - Code Motion

2. **Explain peephole optimization. (8) NOV/DEC2013 ,NOV/DEC2012, MAY/JUNE 2012**

   Contents

   - Write the Reduntant Loads And Stores
   - Write the Unreachable Code
     - Elimination Of Common Subexpressions
     - Elimination Of Dead Code
     - Reduction In Strength
     - Use Of Machine Idioms
   - Getting Better Performance