TRIBHUVAN UNIVERSITY

Institute of Science and Technology

TU QUESTIONS-ANSWERS 2078

Course Title: Compiler Design and Construction — Full Marks: 60 + 20 + 20
Course No: CSC365 — Pass Marks: 24 + 8 + 8
Nature of the Course: Theory + Lab — Credit Hrs: 3
Semester: VI

Section A

Attempt any two questions [2 x 10 =20]

1. List out the tasks performed by lexical analyzer. Define DFA. Convert the Regular Expression (a+b)*a(a+b) to DFA directly. [3+1+6]

Ans: The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis. Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it. Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.

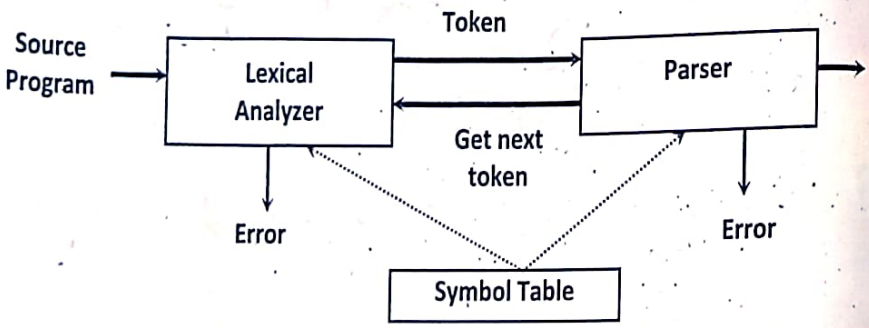


Figure: Place Lexical analyzer in Phases of compiler

Tasks performed by lexical analyzer

- Helps to identify token into the symbol table.
- Removes white spaces and comments from the source program.
- Correlates error messages with the source program.
- Helps you to expand the macros if it is found in the source program.
- Read input characters from the source program.

Deterministic Finite Automaton (DFA)

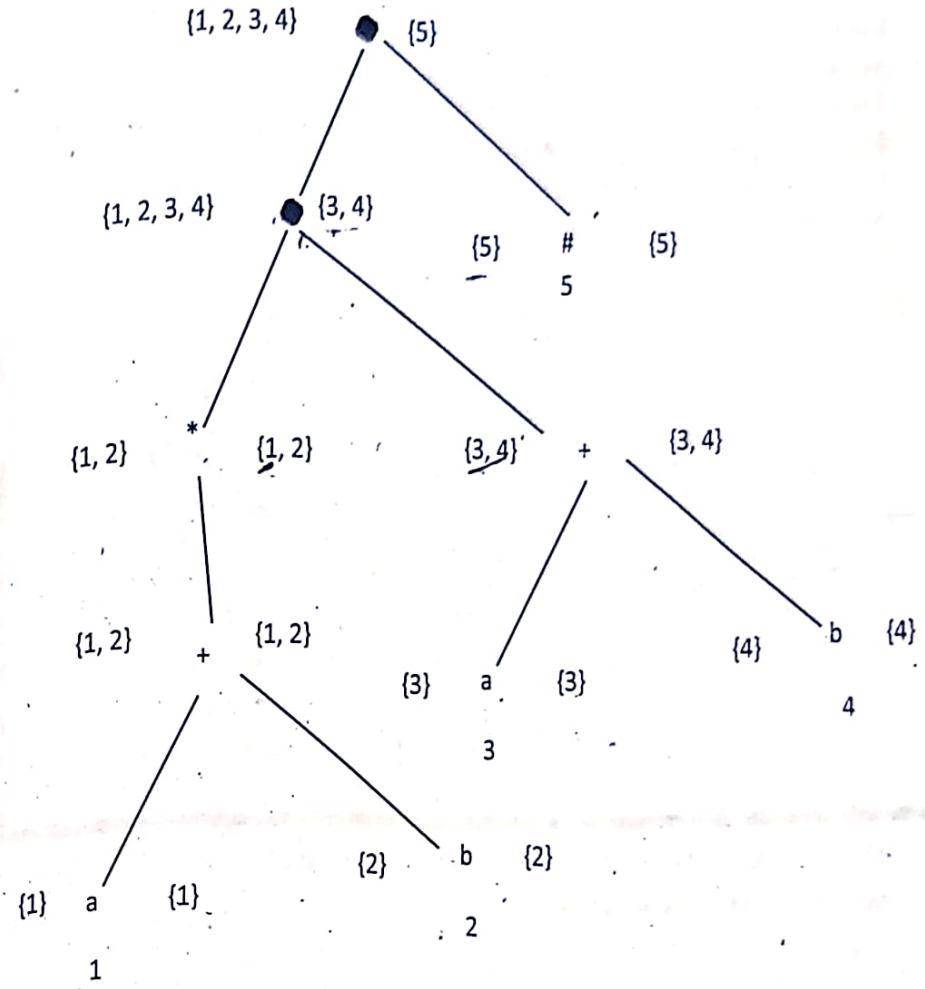
In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

Convert the Regular Expression (a+b)*a(a+b) to DFA directly

Step 1: At first augment the given regular expression as,

(a+b)*.a. (a+b).#

Step 2: Now construct syntax tree of augmented regular expression as,



Step 3: Compute followpos as,

(a+b)*. a. (a+b). #
1 2 3 4 5 6

Followpos(1) = {1, 2, 3}
Followpos(2) = {1, 2, 3}
Followpos(3) = {4, 5}
Followpos(4) = {6}
Followpos(5) = {6}
Followpos(6) = { Φ }

Step 4: After we calculate follow positions, we are ready to create DFA for the regular expression as,

Starting state of DFA = S_1 = Firstpos(Root node of Syntax tree) = {1, 2, 3, 4}

Mark S_1

For a:follwpos(1, 3, 4) = {1, 2, 3, 4, 5, 6} → S_2

For b:followpos(2, 5) = {1, 2, 3, 6} → S_3

Mark S_2

For a:follwpos(1, 3, 4) = {1, 2, 3, 4, 5, 6} → S_2

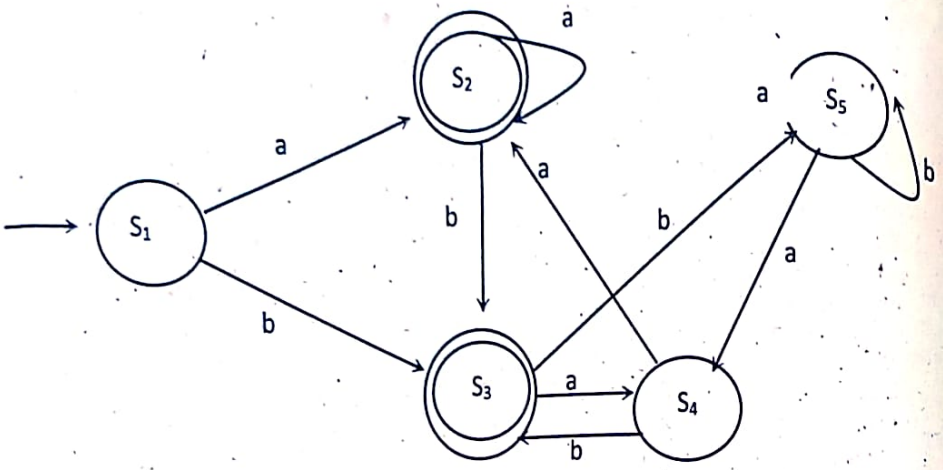
For b:followpos(2, 5) = {1, 2, 3, 6} → S_3

Mark S_3

For a:follwpos(1, 3) = {1, 2, 3, 4, 5} → S_4

For b:followpos(2) = {1, 2, 3} → S_5

Mark S_4
For a:follwpos (1, 3, 4) = {1, 2, 3, 4, 5, 6} → S_2
For b:followpos(2, 5) = {1, 2, 3, 6} → S_3
Mark S_5
For a:follwpos(1, 3) = {1, 2, 3, 4, 5} → S_4
For b:followpos(2) = {1, 2, 3} → S_5
Now there was no new states occur.
So starting state of DFA = {S_2, S_3}



2. **Differentiate between LR(0) and LR(1) algorithm. Construct LR(1) parse table for the following grammar.** [3+7]

$S \rightarrow AA$
$A \rightarrow aA \mid b$

Ans: Difference between LR(0) and LR(1) algorithm

- LR(0)requires just the canonical items for the construction of a parsing table, but LR(1) requires the look-ahead symbol as well.
- LR(1) allows us to choose the correct reductions and allows the use of grammar that are not uniquely invertible while LR(0) does not.
- LR(0) reduces at all items whereas LR(1) reduces only at look-ahead symbols.
- LR(0) = canonical items and LR(1) = LR(0) + look-ahead symbol.
- In canonical LR(0) collection, state/item is in the form:
  E.g. I2 = {C → AB•}

Whereas In cannonicalLR(1) collection, state/item is in the form:
E.g. I2 = {C →d•, e/d} where e &d are look-ahead symbol.

Numerical part,
Construct LR(1) parsing table of following grammar,
$S \rightarrow AA$
$A \rightarrow aA \mid b$
Solution: The augment the given grammar is,
$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

I_0 = Closure ($S' \rightarrow \bullet S$) = {$S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet AA, \$$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b$}

I_1 = Goto (I_0, S) = closure ($S' \rightarrow S\bullet, \$$) = {$S' \rightarrow S\bullet, \$$}

I_2 = Goto (I_0, A) = closure ($S \rightarrow A\bullet A, \$$) = {$S \rightarrow A\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$$}

I_3 = Goto (I_0, a) = Closure ($A \rightarrow a\bullet A, a/b$) = {$A \rightarrow a\bullet A, a/b$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b$}

I_4= Goto (I_0, b) = closure ($A \rightarrow b\bullet, a/b$) = {$A \rightarrow b\bullet, a/b$}

I_5= Goto (I_2, A) = Closure ($S \rightarrow AA\bullet, \$$) = {$S \rightarrow AA\bullet, \$$}

I_6= Goto (I_2, a) = Closure ($A \rightarrow a\bullet A, \$$)
= {$A \rightarrow a\bullet A, \$$
$A \rightarrow \bullet aA, \$$
$A \rightarrow \bullet b, \$$}

I_7= Goto (I_2, b) = Closure ($A \rightarrow b\bullet, \$$) = {$A \rightarrow b\bullet, \$$}

I_8= Goto (I_3, A) = Closure ($A \rightarrow aA\bullet, a/b$) = {$A \rightarrow aA\bullet, a/b$}

Goto (I_3, a) = Closure ($A \rightarrow a\bullet A, a/b$) = {$A \rightarrow a\bullet A, a/b$
$A \rightarrow \bullet aA, a/b$
$A \rightarrow \bullet b, a/b$} same as I_3

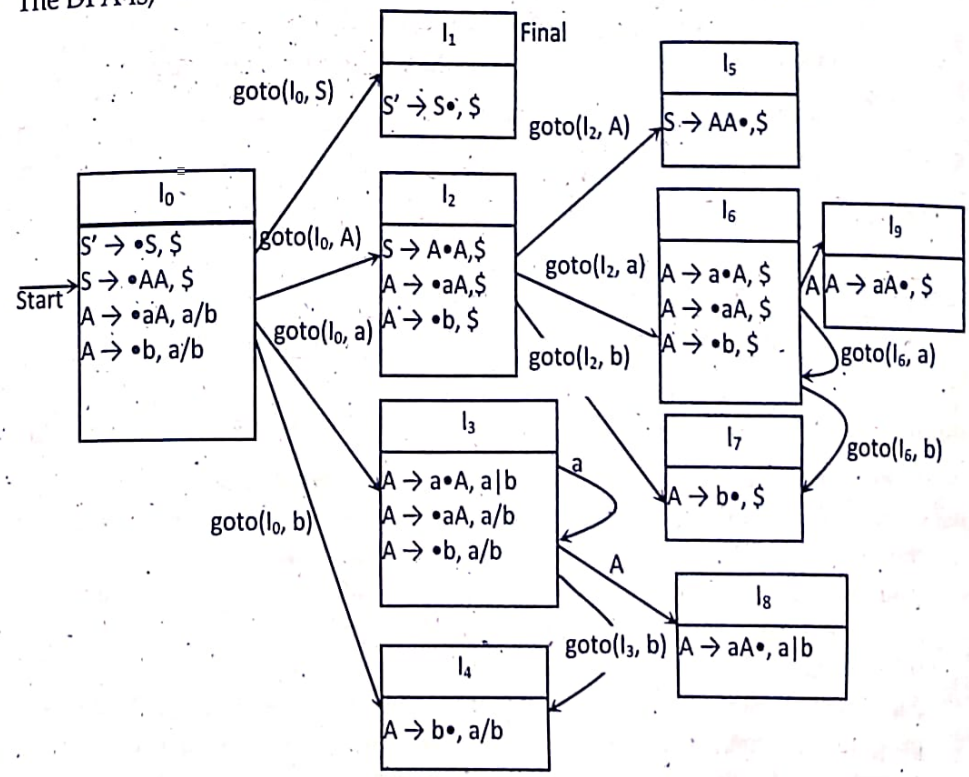
Goto (I_3, b) = Closure ($A \rightarrow b\bullet, a/b$) same as I_4

I_9= Goto (I_6, A) = Closure ($A \rightarrow aA\bullet, \$$) = {$A \rightarrow aA\bullet, \$$}

Goto (I_6, a) = Closure ($A \rightarrow a\bullet A, \$$) same as I_6

Goto (I_6, b) = Closure ($A \rightarrow b\bullet, \$$) same as I_7

The DFA is,



The LR(1) parsing table is,

States	Action table			Goto table	
	a	B	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

3. Define type checking. Differentiate between type casting and coercion. Write SDD to carry out type checking for the following expression. [1+3+6]

E→ n |E*E|E==E |E[E] |E↑

Ans: Compiler must check that the source program follows both the syntactic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables. The design of a type checker for a language is based on information about the syntactic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

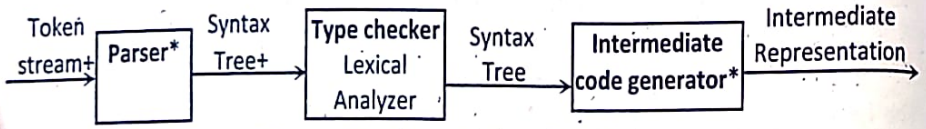


Figure: Position of Type Checker

SDD to carry out type checking for the following expression

E→ n |E*E|E==E |E[E] |E↑

E →n	Print E.val
E → E1 * E2	{E.type = (E1.type == E2.type)? E1.type: type_error}
E → E1 [E2]	{E.type = (E2.type == int and E1.type == array(s, t))? t: type_error}
E → E1 ↑	{E.type = (E1.type == pointer(t)) ? t : type_error}
S → E= = E	{S.type = (E1.type == E2.type)? void: type_error}

Section B

Attempt any EIGHT questions **[8 x 5 = 40]**

4. Define compiler and differentiate it with an interpreter.

Ans: Compiler: A compiler is a translator software program that takes its input in the form of program written in one particular programming language (source language) and produce the output in the form of program in another language (object or target language).

Interpreter

An interpreter is a computer program, which coverts each high-level program statement into the machine code. This includes source code, pre-compiled code, and scripts. Both compiler and interpreters do the same job

which is converting higher level programming language to machine code. However, a compiler will convert the code into machine code (create an exe) before program run. Interpreters convert code into machine code when the program is run.

Difference between Compiler and Interpreter

Basis of difference	Compiler	Interpreter
Programming Steps	• Create the program. • Compile will parse or analyses all of the language statements for its correctness. If incorrect, throws an error • If no error, the compiler will convert source code to machine code. • It links different code files into a runnable program(known as exe) • Run the Program	• Create the Program • No linking of files or machine code generation • Source statements executed line by line DURING Extion
Advantage	The program code is already translated into machine code. Thus, it code execution time is less.	Interpreters are easier to use, especially for beginners.
Disadvantage	You can't change the program without going back to the source code.	Interpreted programs can run on computers that have the corresponding interpreter.
Machine code	Store machine language as machine code on the disk	Not saving machine code at all.
Running time	Compiled code run faster	Interpreted code run slower
Model	It is based on language translation linking-loading model.	It is based on Interpretation Method.
Program generation	Generates output program (in the form of exe) which can be run independently from the original program.	Do not generate output program. So they evaluate the source program at every time during execution.
Execution	Program execution is separate from the compilation. It performed only after the entire output program is compiled.	Program Execution is a part of Interpretation process, so it is performed line by line.
Memory requirement	Target program execute independently and do not require the compiler in the memory.	The interpreter exists in the memory during interpretation.

Basis of difference	Compiler	Interpreter
Best suited for	Bounded to the specific target machine and cannot be ported. C and C++ are a most popular a programming language which uses compilation model.	For web environments, where load times are important. Due to all the exhaustive analysis is done, compiles take relatively larger time to compile even small code that may not be run multiple times. In such cases, interpreters are better.
Code Optimization	The compiler sees the entire code upfront. Hence, they perform lots of optimizations that make code run faster	Interpreters see code line by line, and thus optimizations are not as robust as compilers
Dynamic Typing	Difficult to implement as compilers cannot predict what happens at turn time.	Interpreted languages support Dynamic Typing
Usage	It is best suited for the Production Environment	It is best suited for the program and development environment.
Error execution	Compiler displays all errors and warning at the compilation time. Therefore, you can't run the program without fixing errors	The interpreter reads a single statement and shows the error if any. You must correct the error to interpret next line.
Input	It takes an entire program	It takes a single line of code.
Output	Compliers generates intermediate machine code.	Interpreter never generate any intermediate machine code.
Errors	Display all errors after, compilation, all at the same time.	Displays all errors of each line one by one.
Pertaining Programming languages	C,C++,C#, Java all use complier.	PHP, Perl, Ruby uses an interpreter.

5. What are the typical entries made in symbol table? Explain.

Ans: Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc. It is built in lexical and syntax analysis phases. The information is collected by the analysis phases (front end) of compiler and is used by synthesis phases (back end) of compiler to generate code. It is used by compiler to achieve compile time efficiency.

Symbol Table Entries

Each entry in symbol table is associated with attributes that support compiler in different phases.

Items Stored in Symbol Table:

- Variable names and constants
- Procedure and function names

- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information Used by Compiler from Symbol Table:

- **Name**
  - Name of the identifier
  - May be stored directly or as a pointer to another character string in an associated string table names can be arbitrarily long.
- **Type**
  - Type of the identifier: variable, label, procedure name etc.
  - For variable, its type: basic types, derived types etc.
- **Location**
  - Offset within the program where the identifier is defined
- **Scope**
  - Region of the program where the current definition is valid
- **Other attributes:** array limits, fields of records, parameters, return values etc.

6. **Define left recursive grammar. Remove left recursion from the following grammar.**

$S \rightarrow SB \mid Ca$
$B \rightarrow Bb \mid c$
$C \rightarrow aB \mid a$

Ans: A grammar is **left recursive** if it has a non-terminal A such that there is a derivation.

$A \rightarrow A\alpha$. for some string α

Left recursion causes recursive descent parser to go into infinite loop. Top down parsing techniques cannot handle left - recursive grammars. So, we have to convert our left recursive grammar which is not left recursive. The left recursion may appear in a single derivation called immediate left recursion, or may appear in more than one step of the derivation. So, we have to convert our left-recursive grammar into an equivalent grammar which is non left-recursive.

Remove left recursion from the following grammar.

$S \rightarrow SB \mid Ca$
$B \rightarrow Bb \mid c$
$C \rightarrow aB \mid a$

Solution:

$S \rightarrow SB \mid Ca$	$S \rightarrow Ca\ S'$
	$S' \rightarrow BS' \mid \varepsilon$
$B \rightarrow Bb \mid c$	$B \rightarrow c\ B'$
	$B' \rightarrow bB' \mid \varepsilon$
$C \rightarrow aB \mid a$	

Resulting grammar with left recursive free is,

$S \rightarrow Ca\ S'$
$S' \rightarrow BS' \mid \varepsilon$
$B \rightarrow c\ B'$
$B' \rightarrow bB' \mid \varepsilon$
$C \rightarrow aB \mid a$

7. What are the disadvantages of shift reduce parsing? Perform shift reduce parsing of string w=(x-x)-(x/x) for given grammar

$E \rightarrow E\text{-}E \mid E/E \mid (E) \mid x$

Ans: A shift reduce parser tries to reduce the given input string into the starting symbol. At each reduction step, a substring of the input matching to the right side of a production rule is replaced by non - terminal at the left side of that production rule. If the substring is chosen correctly, the rightmost derivation of that string is created in reverse order. Simply the process of reducing the given input string into the starting symbol is called shift-reduce parsing.

A string ⟶ the starting symbol

Reduced to

Some grammars cannot be parsed using shift-reduce parsing and result in conflicts. There are two kinds of shift-reduce conflicts:

Shift / Reduce Conflict

Here, the parser is not able to decide whether to shift or to reduce.

Example: if $A \rightarrow ab$

$A \rightarrow abcd$, and the stack contains $ab, and the input buffer contains cd$, the parser cannot decide whether to reduce $ab to $A or to shift two more symbols before reducing.

Reduce / Reduce Conflict

Here, the parser cannot decide which sentential form to use for reduction.

For example: if $A \rightarrow bc$ and $B \rightarrow abc$ and the stack contains $abc, the parser cannot decide whether to reduce it to $aA or to $B

Perform shift reduce parsing of string w=(x-x)-(x/x) for given grammar

$E \rightarrow E\text{-}E \mid E/E \mid (E) \mid x$

Stack	Input Buffer	Parsing Action
$	(x-x)-(x/x) $	Shift
$ (	x-x)-(x/x) $	Shift
$ (x	-x)-(x/x) $	Reduce by E → x
$ (E	-x)-(x/x) $	Shift
$ (E -	x)-(x/x) $	Shift
$ (E - x	)-(x/x) $	Reduce by E → x
$ (E - E	)-(x/x) $	Reduce by E → E- E or shift conflict
$ (E	)-(x/x) $	Shift
$ (E)	-(x/x) $	Reduce by E → (E)
$ E	-(x/x) $	Shift
$ E -	(x/x) $	Shift
$ E - (	x/x) $	Shift
$ E - (x	/x) $	Reduce by E → x
$ E - (E	/x) $	Shift
$ E - (E /	x) $	Shift
$ E - (E / x	) $	Reduce by E → x
$ E - (E / E	) $	Reduce by E → E/E
$ E - (E	) $	Shift
$ E - (E)	$	Reduce by E → (E)
$ E - E	$	Reduce by E → E- E
$ E	$	Accept

8. **Define attribute grammar with example of inherited and synthesized attributes.**

Ans: **Synthesized Attribute (↑)**

If the value of the attribute only depends upon its children then it is **synthesized attribute.** Simply the attributes of a node that are derived from its children nodes are called synthesized attributes. Terminals do not have inherited attributes. A non-terminal 'A' can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree. To illustrate, assume the following production:

$S \rightarrow ABC$

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Example for Synthesized Attributes

Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

Production	Semantic Rules
$L \rightarrow E$ return	print(E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	F.val = digit.lexval

Inherited Attribute (→, ↓)

Simply, a node in which attributes are derived from the parent or siblings of the node is called inherited attribute of that node. If the value of the attribute depends upon its parent or siblings then it is inherited attribute. As in the following production,

$S \rightarrow ABC$

'A' can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Example for Inherited Attributes

Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

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

The non-terminal T has a synthesized attribute, type, determined by the keyword in the declaration. The production $D \rightarrow T\ L$ is associated with the semantic rule L.in = T.type which set the inherited attribute L.in.

9. **Define three address codes. Write down Quadruples for**
a = -b * (c+d) /e

Ans: A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three

address statement is known as three address code. Three-address statement is of the general form

$$x = y \text{ op } z$$

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 a sequence

$t_1 = y * z$

$t_2 = x + t_1$

Where, t_1 and t_2 are compiler-generated temporary names.

Write down Quadruples for

a = -b * (c+d) /e

The three address code is,

$t_1 = - b$

$t_2 = c+d$

$t_3 = e$

$t_4 = t_1 * t_2$

$t_5 = t_4 / t_3$

$a = t_5$

Quadruples:

	Op	arg1	arg2	result
(0)	-	b		t_1
(1)	+	c	d	t_2
(2)		e		t_3
(3)	*	t_1	t_2	t_4
(4)	/	t_4	t_3	t_5
(5)	=	t_5		a

10. List out different types of run time storage management techniques. Explain any one of them.

Ans: The information which required during an execution of a procedure is kept in a block of storage called an activation record. "Activation record is a block of memory used for managing information needed by a single execution of a procedure."

Different storage allocation strategies are:

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

The following three-address statements are associated with the run-time allocation and de-allocation of activation records:

1. Call
2. Return
3. Halt
4. Action, a placeholder for other statements

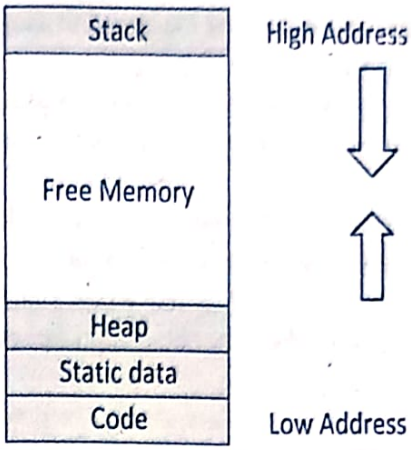


Figure: Storage allocation

Static Storage Allocation

In static allocation, if memory is created at compile time, memory will be created in the static area and only once. It don't support dynamic data structure i.e. memory is created at compile time and de-allocated after program completion.

Disadvantages

- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time.

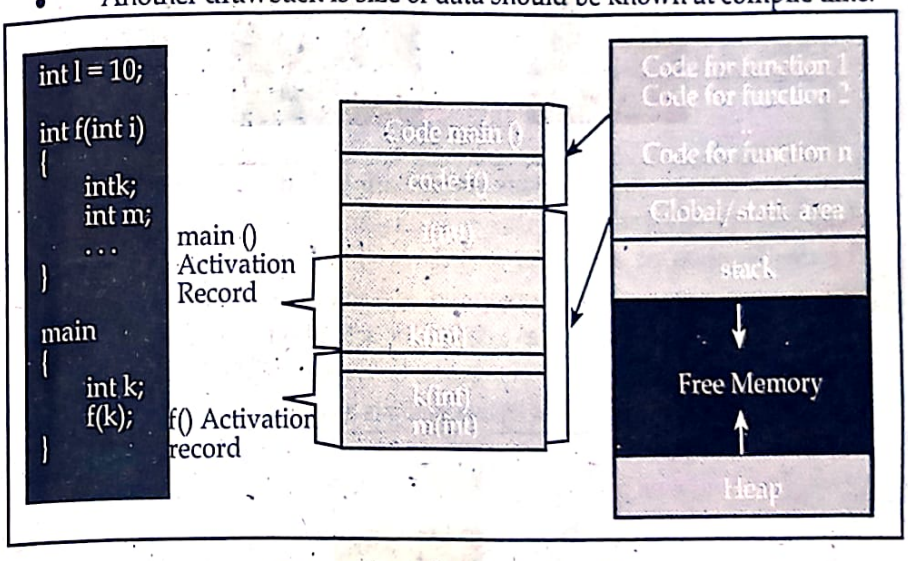


Figure: Static storage allocation

Stack Storage Allocation

Stack allocation is based on the idea of a control stack.

- A stack is a Last In First Out storage device where new storage is allocated and de-allocated at only one "end", called the top of the stack.
- Storage is organized as a stack and activation records are pushed and popped as activations begin and end, respectively.
- Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made.

- Furthermore, the values of local are detected when the activation end That is, the values are lost because the storage for locals disappea when the activation record is popped.
- At run time, an activation record can be allocated and de-allocated b incrementing and decrementing top of the stack respectively.

Advantages

- It supports recursion as memory is always allocated on block entry.
- It allows creating data structure dynamically.
- It allows an array declaration like A(I,J), since actual allocation is mad only at execution time. The dimension bounds need not be known a compile time.

Disadvantages

- Memory addressing can be done using pointers and index registers Hence, this type of allocation is slower than static allocation.

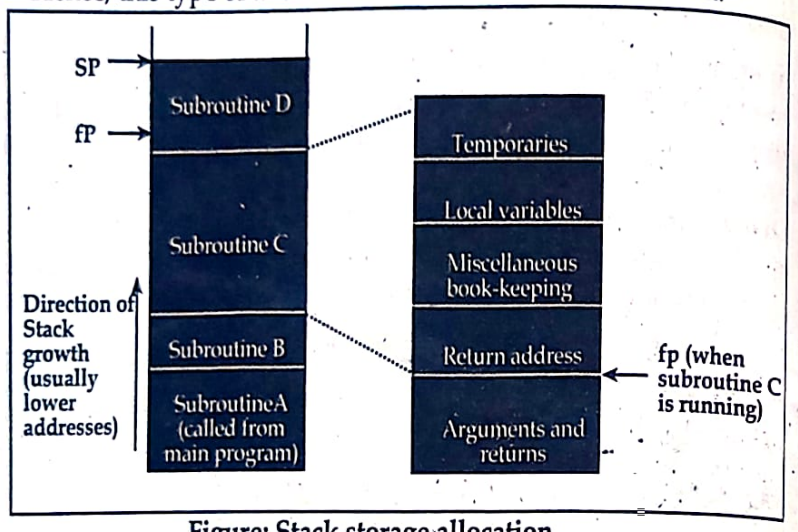


Figure: Stack storage allocation

Heap Storage Allocation

The de-allocation of activation records need not occur in a last-in first-out fashion, so storage cannot be organized as a stack. Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be de-allocated in any order. So over time the heap will consist of alternate areas that are free and in use. Heap is an alternate for stack.

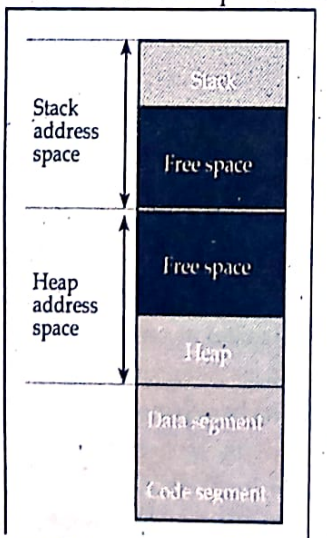


Figure: Heap storage allocation

11. What is the advantage of code optimization? Explain about dead-code elimination.

Ans: 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. The process of code optimization involves:

- Eliminating the unwanted code lines
- Rearranging the statements of the code

The optimized code has the following **advantages:**

1. Optimized code has faster execution speed.
2. Optimized code utilizes the memory efficiency.
3. Optimized code gives better performance.

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.

Dead Code Elimination

In this technique, as the name suggests, it involves eliminating the dead code. The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example:

Code before optimization	Code after optimization

i = 0;
if (i==1)
{
a=x + 5;
}
	i = 0;

12. Explain about the factors affecting target code generation.

Ans: The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained. The different factors affecting target code generation includes:

- Input to code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Evaluation order
- Approaches to code generation issues

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.

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 and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader.
c. **Assembly language:** Code generation is made easier. We can generate symbolic instructions and use macro-facilities of assembler in generating codes.

Memory Management

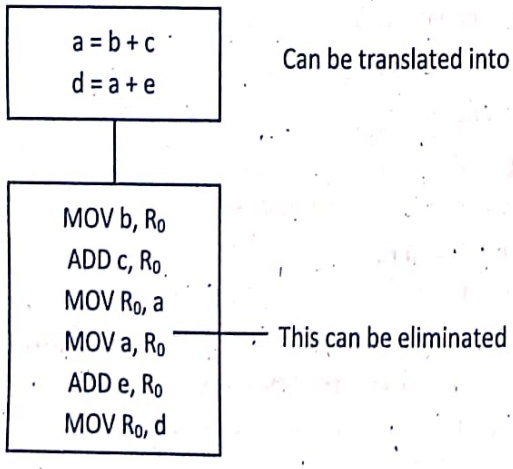
Mapping the names in the source program to addresses of data objects in run-time memory is done by the front end and code generator. It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name. Labels in three-address statements have to be converted to addresses of instructions.

For example, j :gotoi generates jump instruction as follows:

- ifi< j, a backward jump instruction with target address equal to location of code for quadruple i is generated.
- ifi> 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.

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:



Register Allocation

Uses of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers is subdivided into two sub-problems:

- **Register allocation:** the set of variables that will reside in registers at a point in the program is selected.
- **Register assignment**: the specific register is picked to access the variable.

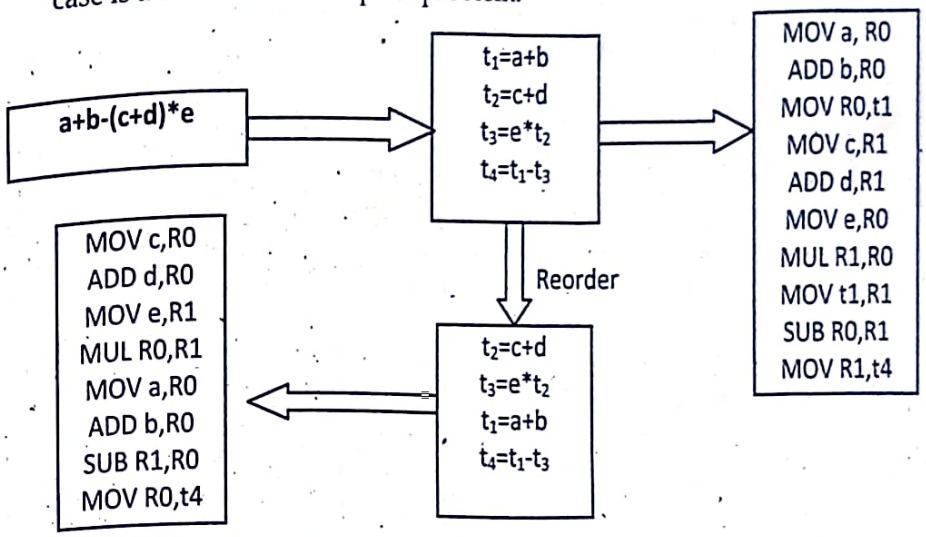
Certain machine requires even-odd **register pairs** for some operands and results. For example, consider the division instruction of the form:

D x, y

Where, x is dividend even registers in even/odd register pair, y is divisor, even register holds the remainder and odd register holds the quotient

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. However, picking the best order in general case is a difficult NP-complete problem.



Approaches to Code Generation

Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Maintainable