A pushdown automaton is:
"Finite state machine" + "a stack"

- A pushdown automaton has three components:

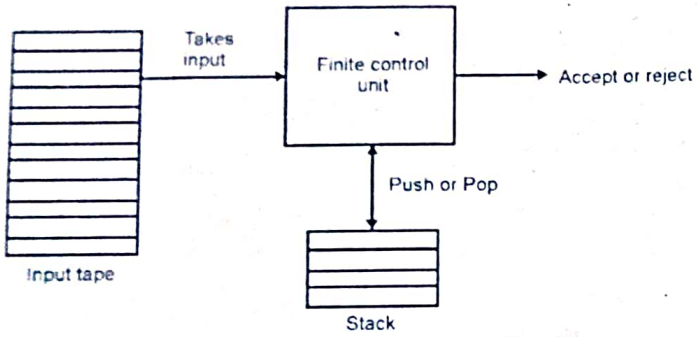
(i) An input tape,
(ii) A control unit, and
(iii) A stack with infinite size.

The stack head scans the top symbol of the stack.

- A stack does two operations:

(i) **Push:** A new symbol is added at the top.
(ii) **Pop:** The top symbol is read and removed.

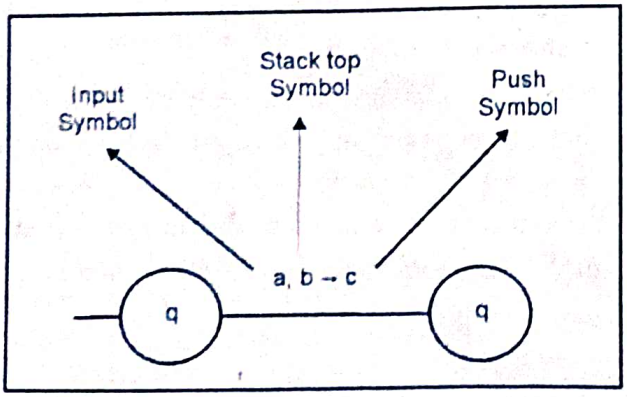
- A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



- A PDA can be formally described as a 7-tuple $(Q, \sum, S, \delta, q_0, I, F)$:

where
Q - The finite number of states
$\sum$ - Input alphabet
S - Stack symbols
δ - The transition function:
$Q \times (\sum \cup \{\varepsilon\}) \times S \times Q \times S^*$
q_0 - The initial state $(q_0 \in Q)$
I - The initial stack top symbol $(I \in S)$
F - A set of accepting states $(F \in Q)$

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as $a,b \rightarrow c$:



This means at state q_1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q_2.

Terminologies Related to PDA:

(i) **Instantaneous Description:**

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where
q - is the state
w - is unconsumed input
s - is the stack contents

(ii) **Turnstile Notation:**

- The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol " $\vdash$ ".
- Consider a PDA $(Q, \sum, S, \delta, q_0, I, F)$. A transition can be mathematically represented by the following turnstile notation:
  $(p, aw, T\beta) \vdash (q, w, \alpha b)$
- This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed and the top of the stack 'T' is replaced by a new string 'α'.

Note: If we want zero or more moves of a PDA, we have to use the symbol ($\vdash$ *) for it.

UNIT - III

SEMANTIC ANALYSIS: ATTRIBUTE GRAMMER

Q.6. Explain Semantic Analysis.

Ans. Semantic Analysis:

- Semantic analysis is the third phase of Compiler. Semantic analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

- Type checking is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

Semantic Analyzer:

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors:

Errors recognized by semantic analyzer are as follows:

(a) Type mismatch
(b) Undeclared variables
(c) Reserved identifier misuse

Functions of Semantic Analysis:

(1) **Type Checking:**
Ensures that data types are used in a way consistent with their definition.

(2) **Label Checking:**
A program should contain labels references.

(3) **Flow Control Check:**
Keeps a check that control structures are used in a proper manner. (example: no break statement outside a loop).

Example:

float x = 10.1;
float y = x*30;

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

Static and Dynamic Semantics:

(1) **Static Semantics:**
It is named so because of the fact that these are checked at compile time. The static semantics and meaning of program during execution, are indirectly related.

(2) **Dynamic Semantic Analysis:**
It defines the meaning of different units of program like expressions and statements. These are checked at runtime unlike static semantics.

RUN-TIME ENVIRONMENT, PROCEDURE ACTIVATION

Q.7. Explain Run-Time Environment.

Ans. Run-Time Environment:

- A Program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.
- By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.
- Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Q.8. Explain Activation Trees.

OR Explain Procedure Activation.

Ans. Activation Trees:

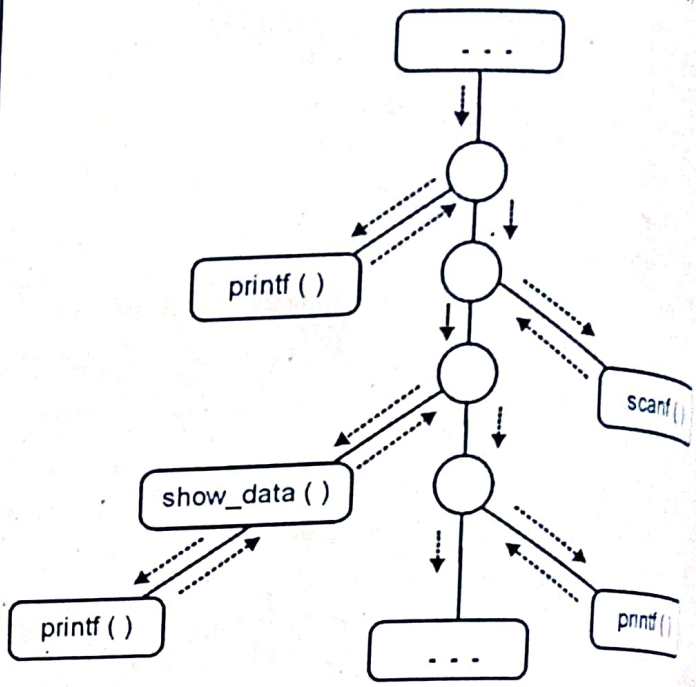
- A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.
- The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e. parameters which are used to send input to the called procedure.
Return Value	Stores return values.

- Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.
- We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.
- To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue...");
. . .
int show_data(char *user)
{
    printf("Your name is %s", username);
    return 0;
}
. . .
```

- Below is the activation tree of the code given



- The procedures are executed in depth-fir manner, thus stack allocation is the best suitab form of storage for procedure activations.

PARAMETER PASSING, VALUE RETURN

Q.9. Explain Parameter Passing.

Ans. Parameter Passing:

The communication medium among procedures known as parameter passing. The values of variables from a calling procedure are transfer to the called procedure by some mechanis Before moving ahead, first go through some ba terminologies pertaining to the values in program.

(i) r-value:

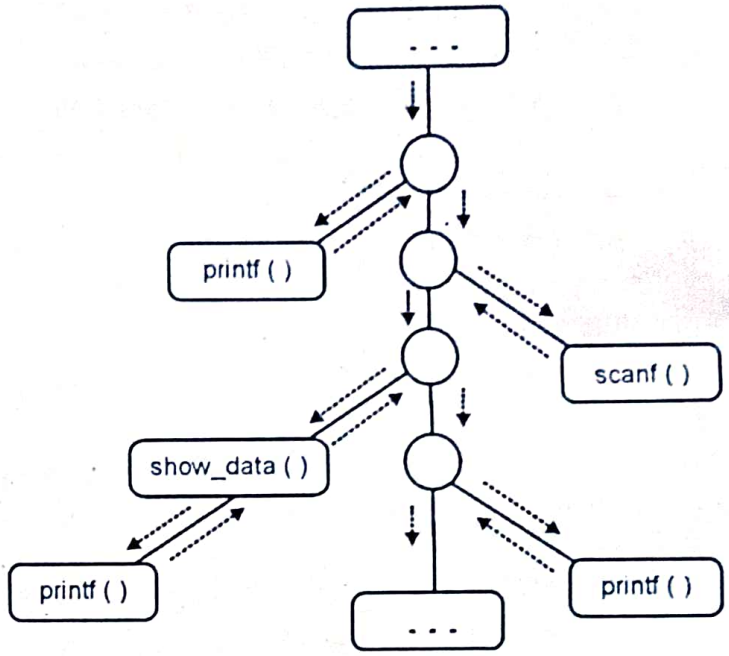
The value of an expression is called its r-val The value contained in a single variable b becomes an r-value if it appears on the right-h side of the assignment operator. r-values always be assigned to some other variable.

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e. parameters which are used to send input to the called procedure.
Return Value	Stores return values.

- Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.
- We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.
- To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue...");
. . .
int show_data(char *user)
{
    printf("Your name is %s", username);
    return 0;
}
. . .
```

- Below is the activation tree of the code given.



- The procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

PARAMETER PASSING, VALUE RETURN

Q.9. Explain Parameter Passing.

Ans. Parameter Passing:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

(i) r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

This unit develops the theme of the translation of languages guided by context free grammars we associate information with a programming language construct by attaching attributes to the grammar. Symbols representing the construct values for attributes are computed by semantic rules associated with the grammar productions.

There are notational framework for intermediate code generation that is an extension of context-free grammars. This framework is called as syntax-directed translation scheme. It allows subroutines or 'semantic actions' to be attached to the productions of context free grammar. These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.

We will see what are inherited, synthesized attribute, evaluation of attributes, L and S attributed definition, implementation of SDTS, evaluation of expression using semantic actions and finally the postfix, syntax tree and three address code used for intermediate code representation.

SPECIFICATION OF TRANSLATIONS

Q.1. Write short note on specification of translation.

Ans. Specification translation :

- It involves specifying what the construct is, as well as specifying the translating rules for construct.
- Whenever a compiler encounters that construct in a program, it will translate the construct according to the rules of translation.
- Translation does not necessarily mean generating either intermediate code or object code.
- Translation also involves adding information into the symbol table as well as performing construct specific computations.
- For example, if a construct is a declarative about the constructs type attribute into symbol table.
- If construct is an expression then its translation generates the code for evaluating the expression.
- Translation of a construct involves manipulating the values of various quantities.

When translating the declarative statement int a, b, c, the compiler need to extract the type int and add it to the symbol table records of a, b, c.

- This require compiler to keep the track of the type int as well as the pointers to the symbol records containing a, b and c.
- Syntax directed definitions use CFG to specify the syntactic structure of the construct.
- It associates a set of attributes with each grammar symbol and with each production, it associates a set of semantic rules for computing the values of the attributes of the grammar symbols appearing in that production.
- Therefore the grammar and the set of semantic rules constitute syntax directed definitions.

SYNTAX DIRECTED DEFINITION

Q.2. What is syntax-directed definitions?

Ans. Syntax directed definition :

- A 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 the synthesized and inherited attributes of that grammar symbol.
- If we think of a node for the grammar symbol in a parse tree as a record with fields for holding information, then an attribute corresponds to the name of a field.

Q.3. Explain form of a Syntax-directed definition.

Ans.

- In a Syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(C_1, C_2, \ldots\ldots C_k)$, where f is a function and either,

(1) b is a synthesized attribute of A and $C_1, C_2, \ldots\ldots C_k$ are attributes belonging to the grammar symbols of the production, or

(2) b is a inherited attribute of one of the grammar symbols on the right side of the production and $C_1, C_2, \ldots\ldots C_k$ are attributes belonging to the grammar symbol of the production.

- An attribute grammar is a syntax-directed definition in which the function in semantic rules cannot have side effects.

Q.4. Explain semantic action. **CS : S-10(2M), W-10(1M)**

Ans. Semantic action :

- A compiler has to do more than just recognize if a sequence of characters forms a valid sentence in the language.
- The semantic action of a parser perform useful operation
  (1) Build an abstract parse tree.
  (2) Type checking.
  (3) Evaluation in the case of an interpreter.
  (4) Begin the translation process in the case of a compiler.
- In recursive descent parser the semantic action are mixed in with the control flow of parsing.
- In some compiler constructors such as Java CC and Yacc the semantic actions are attached to the production rules.
- In other compiler constructions sable CC for example the syntax tree automatically generated.
- Each symbol, terminal or non terminal, may have its own type of semantic value.

  exp → INT

  exp → exp PLUS exp.

  exp → exp MINUS exp.

  exp → exp MUL exp.

  exp → exp DIV exp.

  exp → MINUS exp.

SYNTAX DIRECTED TRANSLATION SCHEME

Q.5. What is Syntax directed translation scheme? Justify the necessity for intermediate code and give some example.

Ans. Syntax directed translation scheme :

- A syntax directed translation scheme is a notational framework for intermediate code generation which is an extension of context-free grammars.
- It allows subroutines or "Semantic action" to be attached to the productions of a context-free grammar.
- These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.
- Enable the compiler designer to express the generation of code directly in terms of the syntactic of the source language.

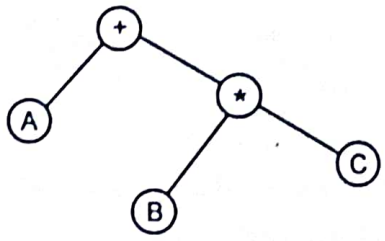
Necessity for the intermediate code. :

- Syntax directed translation scheme can be used for translation of source code to target code, which generally speeds up the compiler, the optimization of machine or assembly code is harder. Hence some form of intermediate code is generated on which optimization can be performed easily.
- The reason why efficient machine or assembly code is hard to generate is that one is immediately forced to choose a particular register to hold result difficult. Usually the intermediate code is choosen in such a way like assembly language each statement involves at most one operation, but unlike assembly code, the register in which each operation occurs is left unspecified.

Example of intermediate code :

(1) **Postfix notation :** abc* - is a postfix notation for a + b*c.

(2) **Syntax tree :** For a + b*c syntax tree is shown below.



(3) **Three address code :** The three address code for a + b*c will be

T_1 : b * c

T_2 : a + T_1 , where T1 and T2 are compiler generated temporaries.

INHERITED AND SYNTHESIZED ATTRIBUTES

Q.6. What are the implementations of the translations specified by syntax-directed definitions?

OR What are synthesized and inherited attributes?

OR Define synthesized and inherited attribute. **CS : W-10(2M)**

Ans.

- Attributes are associated with the grammar symbols that are the labels of the parse tree node.
- When a semantic rule is evaluated, the parser computes the value of an attribute at a parse tree node.
- To evaluate the semantic rules and carry out translation, the parse tree must be traversed and the values of the attributes at the nodes computed.
- The order in which we traverse the parse tree nodes depends on the dependencies of the attributes at the parse tree node.

Q.4. Explain semantic action. **CS : S-10(2M), W-10(1M)**

Ans. Semantic action :

- A compiler has to do more than just recognize if a sequence of characters forms a valid sentence in the language.
- The semantic action of a parser perform useful operation
  (1) Build an abstract parse tree.
  (2) Type checking.
  (3) Evaluation in the case of an interpreter.
  (4) Begin the translation process in the case of a compiler.
- In recursive descent parser the semantic action are mixed in with the control flow of parsing.
- In some compiler constructors such as Java CC and Yacc the semantic actions are attached to the production rules.
- In other compiler constructions sable CC for example the syntax tree automatically generated.
- Each symbol, terminal or non terminal, may have its own type of semantic value.

  exp → INT

  exp → exp PLUS exp.

  exp → exp MINUS exp.

  exp → exp MUL exp.

  exp → exp DIV exp.

  exp → MINUS exp.

SYNTAX DIRECTED TRANSLATION SCIIEME

Q.5. What is Syntax directed translation scheme? Justify the necessity for intermediate code and give some example.

Ans. Syntax directed translation scheme :

- A syntax directed translation scheme is a notational framework for intermediate code generation which is an extension of context-free grammars.
- It allows subroutines or "Semantic action" to be attached to the productions of a context-free grammar.
- These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.
- Enable the compiler designer to express the generation of code directly in terms of the syntactic of the source language.

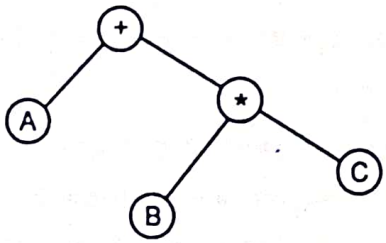
Necessity for the intermediate code. :

- Syntax directed translation scheme can be used for translation of source code to target code, which generally speeds up the compiler, the optimization of machine or assembly code is harder. Hence some form of intermediate code is generated on which optimization can be performed easily.
- The reason why efficient machine or assembly code is hard to generate is that one is immediately forced to choose a particular register to hold result difficult. Usually the intermediate code is choosen in such a way like assembly language each statement involves at most one operation, but unlike assembly code, the register in which each operation occurs is left unspecified.

Example of intermediate code :

(1) **Postfix notation :** abc* - is a postfix notation for a + b*c.

(2) **Syntax tree :** For a + b*c syntax tree is shown below.



(3) **Three address code :** The three address code for a + b*c will be

T_1 : b * c

T_2 : a + T_1 , where T1 and T2 are compiler generated temporaries.

INIIERITED AND SYNTIIESIZED ATTRIBUTES

Q.6. What are the implementations of the translations specified by syntax-directed definitions?

OR What are synthesized and inherited attributes?

OR Define synthesized and inherited attribute. **CS : W-10(2M)**

Ans.

- Attributes are associated with the grammar symbols that are the labels of the parse tree node.
- When a semantic rule is evaluated, the parser computes the value of an attribute at a parse tree node.
- To evaluate the semantic rules and carry out translation, the parse tree must be traversed and the values of the attributes at the nodes computed.
- The order in which we traverse the parse tree nodes depends on the dependencies of the attributes at the parse tree node.

- Carrying out the translation specified by the syntax-directed definition involves :

(1) Generating the parse tree for the input string w.

(2) Finding out the traversal order of the parse tree nodes by generating a dependency graph and doing a topological sort of that graph.

(3) Traversing the parse tree in the proper order and getting the semantic rules evaluated.

Synthesized and inherited attributes :

- The attributes associated with a grammar symbols are classified into two types :

(A) Synthesized attributes.

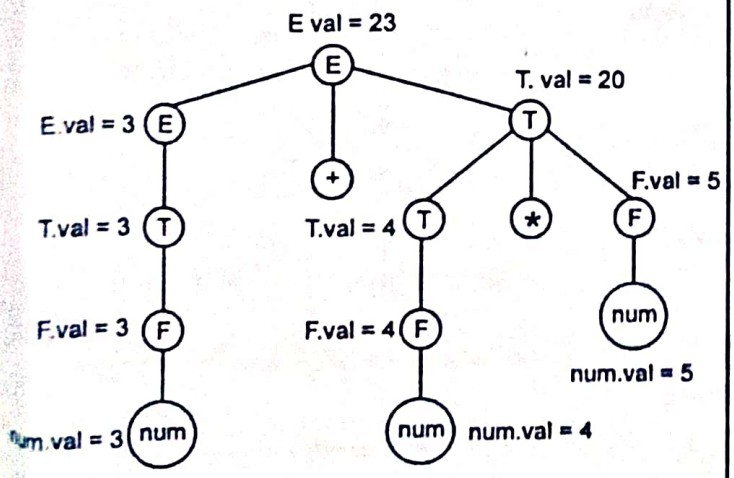
(B) Inherited attributes.

(A) Synthesized attributes :

- If value of attribute at parse tree node is determined from attribute values of the children node, then attribute is defined as synthesized attribute.
- They are evaluated during a single bottom up traversal of parse tree and are as follows :

$E \rightarrow E_1 + T$	E. val : = E_1 . val + T. val
$E \rightarrow T$	E. val : = T. val
$E \rightarrow T_1 * F$	T. val : = T_1 . val * F. val
$T \rightarrow F$	T. val : = F. val
$F \rightarrow id$	F. val : = num. lexval

- The above shown SDTS using synthesized attribute specify the translations, that are needed to be carried by expression evaluator.
- For example, parse tree for an expression 3 + 4 * 5 along with values of attribute at the nodes of parse tree is shown below.



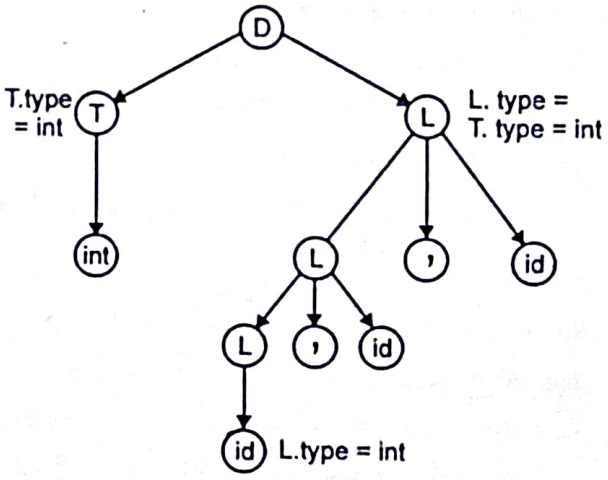
- Syntax directed definitions that use synthesized attributes only are known as "S-attributed" definitions.

(B) Inherited attributes :

- They are those initial values at a node in the parse tree which are defined in terms of attributes of the parent and / or siblings of that node.
- For example,

$D \rightarrow TL$	L. type = T. type
$T \rightarrow int$	T. type = int
$T \rightarrow real$	T. type = real
$L \rightarrow L_1, id$	L_1 . type = L. type
	enter (id . prt, L . type)
$L \rightarrow id$	enter (id . prt, L . type)

- A parse tree for an input string int id_1, id_2, id_3 along with values of the attributes at the nodes of parse tree is as shown below.



- Inherited attributes are convenient for expressing the dependency of a programming language construct on the context in which it appears when inherited attributes are used, then the inter dependencies among the attributes at the nodes of the parse tree must be taken into account when evaluating their semantic rules.
- The interdependencies among attributes are depicted by a directed graph called a "dependency graph".
- If the semantic rule is in the form of a procedure call fun ($a_1, a_2, a_3, ..., a_k$), then it can be transformed into form b = fun ($a_1, a_2, a_3, ..., a_k$), where b is a dummy synthesized attribute.

DEPENDENCY GRAPHS

Q.7. Write short note on dependency graphs.

Ans. Dependency graph :

- If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c
- The interdependencies among the inherited and synthesized attributes at that nodes in a parse tree can be depicted by a directed graph, called a dependency graph
- Before constructing a dependency graph for a parse tree, we put each semantic rule into the form b : = $f(c_1, c_2, ...c_k)$ by introducing a dummy synthesized attribute b for each semantic rule that consists of a procedure call.
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.
- The algorithm for construction of dependency graph for a given parse tree is as follows :

  for each node n in the parse tree do

  for each attribute a of the grammar symbol at n do

  construct a node in the dependency graph for a :

  for each node n in the parse tree do

  for each semantic rule b : = $f(c_1, c_2, ...c_k)$ associated with the production used at n do

  for i = 1 to k do

  construct an edge from the node for ci to the node for b :
- For example, fig. (c) shows the dependency graph for the parse tree in fig. (b). nodes in the dependency graphs are marked by numbers.
- The parse tree in fig (b) is constructed from fig (a).

Production	Semantic rules
D → TL	L. in : = T. type
T → int	T. type : = integer
T → real	T. type ; = real
L → L_1 , id	L_1 . in : = L. in addtype (id. Entry, L. in)
L → id	addtype (id. Entry, L. in)

Fig. (a) Syntax-directed definition with inherited attribute L. in.

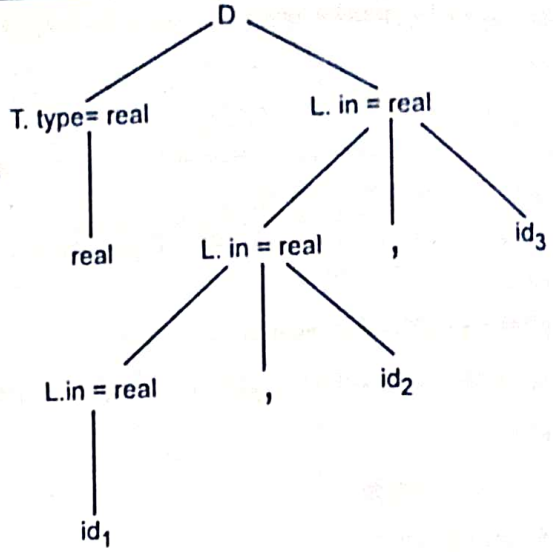


Fig. (b) Parse tree with inherited attribute in at each node labelled L.

- In fig. (c) , there is an edge to node 5 for L in from node 4 for T-type because the inherited attribute L in depends on the attribute T.type according to the semantic rule L. in : = T. type for the production D → TL.
- The two downward edges into nodes 7 and 9 arise, because L_1 in depends on L. in according to the semantic rule L_1 . in = L. in for the production L → L_1 , id. Each of the semantic rules addtype (id. entry, L. in) associated with the L-production leads to the creation of a dummy attribute.
- Nodes 6, 8 and 10 are constructed for these dummy attributes.

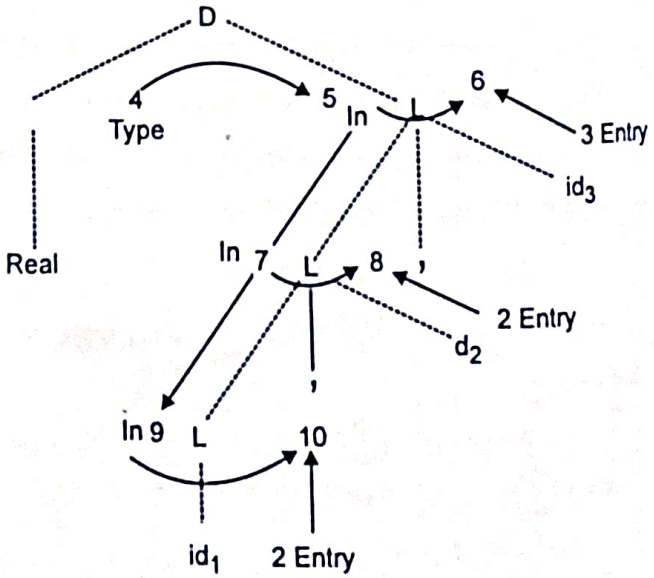


Fig. (c) Dependency graph for parse tree of fig. (b)

Q.8. Explain the term 'topological sort'.

Ans. Topological sort :

A topological sort of a directed cyclic graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes that is, 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.

Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.

That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots, c_k$ in a semantic rule $b := f(c_1, c_2, \ldots, c_k)$ are available at a node before f is evaluated.

For example, each of the edges in the dependency graph in figure goes from a lower-numbered node to a higher-numbered node.

Hence a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers.

From this topological sort, we obtain the following program. We write a_n for the attribute associated with the node numbered n in the dependency graph.

a_4 = real;

$a_5 = a_4$;

addtype (id_3 entry, a_5);

$a_7 = a_5$

$a_9 = a_7$;

addtype (id_1 Entry, a_9);

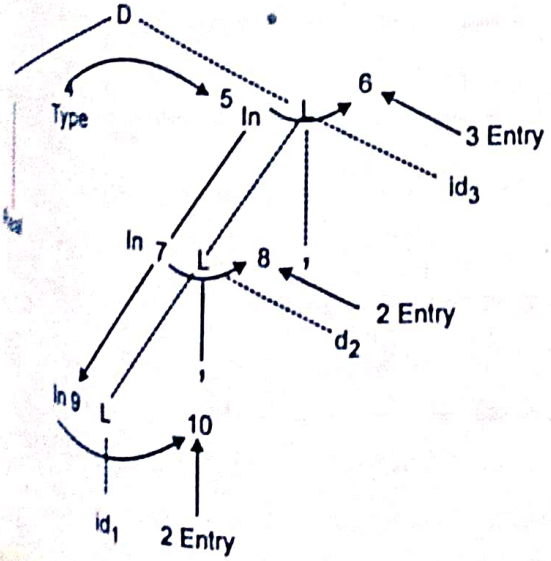


Fig. (d) Dependency graph

EVALUATION ORDER

Q.9. Write a short note on evaluation order.

Ans. Evaluation order :

- A topological sort of a directed acycle graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- 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.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots, c_k$ in a semantic rule $b := f(c_1, c_2, \ldots, c_k)$ are available at a node before f is evaluated.
- From a topological sort of the dependency graph we obtain an evaluation order for the semantic rules.
- Evaluation of the semantic rules in this order yield the translation of the input string.
- The several methods for evaluating semantic rules are as follows :

(1) Parse-tree method :

- At compile time, these method obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input.
- These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle.

(2) Rule-based method :

- At compiler-construction time, the semantic rules associated with productions are analyzed, either by hand or by a specialized tool.
- For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler construction time.

(3) Oblivious method :

- An evaluation order is chosen without considering the semantic rule.
- For example, if translation takes place during parsing than the order of evaluation is forced by the parsing method, independent of the semantic rules.
- An oblivious evaluation order restricts the class of syntax-directed definitions that can be implemented.

VRD

- Rule based an oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.
- A syntax-directed definition is said to be circular if the dependency graph for some parse tree generated by its grammar as a cycle.

TOP DOWN AND BOTTOM UP EVALUATION

Q.10. What is top down and bottom up evaluation of attributes?

Ans.

- A syntax analyzer is a program that perform syntax analysis.
- A parser obtain a string of tokens from lexical analyzer and verifies whether or not string is a valid construct of the source language i.e. whether or not it can be generated by the grammar for the source language.
- The parser either attempts to derive the string of tokens w from the start symbol S or it attempts to reduce w to the start symbol of grammar by tracing the derivations of w in reverse.
- An attempt to derive w from the grammar's start symbol S is equivalent to an attempt to construct the top-down parse tree i.e. it starts from the root node and proceeds towards the leaves.
- Similarly an attempt to reduce w to the grammar's start symbol S is equivalent to an attempt to construct bottom up tree.

Alphabet : An alphabet is a finite set of symbols denoted by symbol Σ.

Language :

- A language is a set of strings formed by using the symbols belonging to some previously chosen alphabet.
- For example, if $\Sigma = \{0, 1\}$ then one of the languages that can be defined over this Σ will be L : { ϵ , 0, 00, 000, 1, 11, 111,...).

TOP DOWN PARSING

Q.11. What is top down parsing?

Ans. Top down parsing :

- Top down parsing attempts to find the left most derivation for an input string w, which is equivalent to constructing a parse tree for the input string w that starts from the root and creates the nodes of the parse tree in predefined order.
- The reason that top-down parsing seeks the left most derivations for an input string w and not the right most derivations is that input string w is scanned by the parser from left to right.
- Since top down parsing attempts to find the leftmost derivations for an input string w, a top down parser may require backtracking because in the attempt to obtain the left most derivation of input string w, a parser may encounter a situation in which a non-terminal A is required to be derived next and there are multiple A productions, such as $A \rightarrow \alpha_1 | \alpha_2 | .... | \alpha_n$.
- In such a situation deciding which A production to use for derivation of A is a problem.
- So, the parser will select one of the A-production to derive A and if this derivation finally leads to the derivation of w, then parser announces the successful completion of parsing.
- For example, consider top-down parser for following grammar

  $S \rightarrow aAb$

  $A \rightarrow cd / c$
- Let input string be w = acb the parser initially creates a tree consisting of a single node, labeled S and input pointer points to a, the first symbol of input string w.
- The parser then user the S-production $S \rightarrow aAb$ to expand tree shown in Fig. (a).

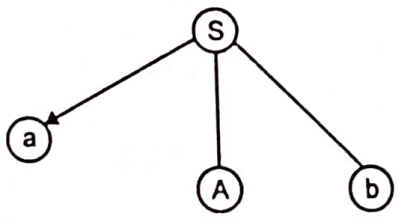


Fig. (a)

- The leftmost leaf labeled a, matches the first input symbol of w. Hence parser will now advance the input pointer to c, the second symbol of string w and consider the next leaf labeled A.
- It will then expand A using first alternative for A in order to obtain the tree as shown in Fig. (b)

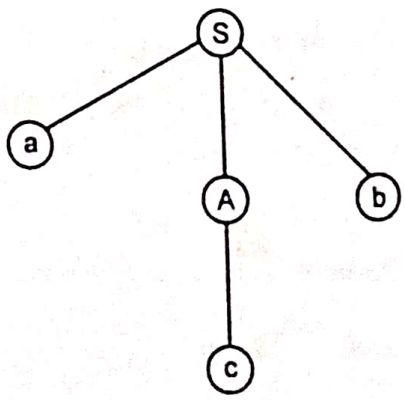


Fig.(b)

The parser now has the match for the second input symbol, so it advances the pointer to b, the third symbol of w and compare it to the label of the next leaf. If the label does not match d, it reports failure and goes back (backtracks) to A as shown in Fig. (c)

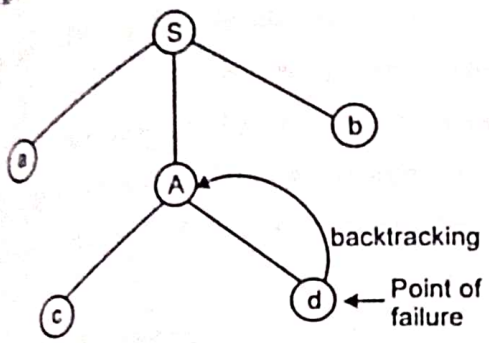


Fig.(c)

- The parser now reset the input pointer to the second input symbol - the position it had when the parser encountered A and it will try a second alternative for A in order to obtain the tree.
- If the leaf c matches the second symbol, if next leaf b matches the third symbol of w, then the parser will halt and announce the successful completion of parsing.

BOTTOM UP PARSING

Q.12. What is bottom up parsing?

Ans. Bottom up parsing :

- This parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the rightmost derivations of w in reverse.
- This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding toward the root i.e. attempting to construct the parse tree from bottom up.
- This involves searching for the substring that matches the right side of any of the productions of the grammar.
- This substring is replaced by the lefthand side nonterminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the right most derivation.
- This process of replacing the right side of the production by the left side nonterminal is called "reduction".
- Hence, reduction is nothing more than performing derivations in reverse.
- The reason why bottom up parsing tries to trace out the right most derivations of an input string w in reverse and not the left most derivation is because the parse scans the input string w from left to right, one symbol per token at a time.
- And to trace out right most derivations of an input string w in reverse, the token of w must be made available in a left to right order.
- For example, if the rightmost derivation sequence of some w is

  $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow ..... \rightarrow \alpha_{n-1} \rightarrow w$

  then bottom up parse starts with w and searches for the occurrence of a substring of w that matches the right side of some production $A \rightarrow \beta$ such that the replacement of β by A will lead to the generation of α_{n-1}.
- The parser replaces β by A, then it searches for the occurrence of a substring at α_{n-1} that matches the right side of some productions $\beta \rightarrow r$ such that replacement of r by β will lead to the generation of α_{n-2}.
- This process continues until the entire w substring is reduced to S, or until the parser encounters an error.
- Bottom up parsing involves the selection of a substring that matches the right side of the production.
- These reduction to the nonterminal on the left side of the production represents one step along the reverse of a right most derivation.
- It leads to the generation of the previous right most derivation.
- This means that selecting a substring that matches the right side of production is not enough ; the positions of this substring in the sentential form is also important.

A handle of a right sentential form :

(1) A handle of a right sentential form r is a production $A \rightarrow \beta$ and a position of β in r.

(2) The string β will be found and replaced by A to produce the priors right sentential form in the right most derivation of r.

(3) If $S \Rightarrow \alpha A\beta \rightarrow \alpha\tau\beta$, then $A \rightarrow \tau$ is a handle of $\alpha\tau\beta$, in the position following α.

(4) For example, consider the grammar :

$E \rightarrow E + E / E * E / id$

and the right most derivation :

$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * id \rightarrow E + id * id \rightarrow id + id * id$

The handles of the sentential forms occurring in the above derivations are shown in table below

Sentential form	Handle
id + id * id	E → id at position preceding +.
E + id * id	E → id at position following +
E → E * id	E → id at position following *
E + E * id	E → E * E at position following +
E + E	E → E + E at position preceding the end marker.

- So, the bottom up parsing is an attempt to detect the handle of a right sentential form.
- This is equivalent to performing right most derivation in reverse and is called "handle pruning".
- So if the right most derivation sequence of some w is $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow ..... \rightarrow \alpha_{n-1} \rightarrow w$, then handle pruning starts with w, the n^{th} right sentential form, the handle $A_n \rightarrow B_n$ of w is located and β_n is replaced by the left side of production $A_n \rightarrow B_n$ in order to obtain α_{n-1}.

L AND S ATTRIBUTE DEFINITIONS

Q.13. Explain the term S-attributed definition and L-attributed definition.

OR Explain L-attributed definitions S-attributed definition.

CS : S-10(2M)

Ans. S-attributed definition :

- A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition. A parse tree showing the values of attributes at each node is called an annotated parse tree.
- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed.
- The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack.
- Whenever, a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.
- A translator for an S-attributed definition can often be implemented with the help of an LR-parser generator.
- From an S-attributed definition, the parser generator can construct a translator that evaluates attributes as it parses the input.
- A bottom-up parser uses a stack to hold information about sub-trees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes.

L - attributed definition :

- An inherited attribute is one whose value at a node in a parser tree is defined in terms of attributes at the parent and / or siblings of that node.
- Inherited attributes are convenient for expressing the dependence of a programming language, constructed on the context in which it appears.
- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1, X_2, ...X_3$ depends only on

(1) The attributes of the symbols $X_1, X, ...X_{j-1}$ to the left of X_j in the production, and

(2) The inherited attributes of A.

$D \rightarrow TL$	L . Type = T . Type
$T \rightarrow$ int	T . type = int
$T \rightarrow$ real	T . type = real
$L \rightarrow L_1$. id	L_1 . type = L . type ; enter (id . prt, L . type) ;
$L \rightarrow$ id	enter (id . ptr, L . type)

- The syntax directed definition given above is an example of the L – attributed definition, because the inherited attribute L . type depends on T. type, and T is to the left of L in the production $D \rightarrow TL$.
- Similarly the inherited attribute L_1 . type depends on the inherited attribute L . type, and L is parent of L_1 in the production $L \rightarrow L_1$, id .
- Note that every S-attributed definition is L-attributed.

IMPLEMENTATION OF SDTS

Q.14. Write short note on implementation of SDTS.

Ans. Implementation of SDTS :

- A syntax-directed translation scheme is a convenient description of what we would like done.
- The output defined is independent of the kind of parser used to construct the parse tree or the kind of mechanism used to compute the translations.
- Thus, a syntax-directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation.
- Another convenience of the approach is that it is easy to modify.
- New productions and semantic actions can often be added without disturbing the existing translations being computed.
- Having written a syntax-directed translation scheme, our next task is to convert it into a program that implements the input-output mapping described.

Mechanism to implement syntax directed translation :

- A useful requirement is that we have a bottom-up parser for the grammar. Some type of LR (1) parser would be quite adequate for most schemes.
- However, we must augment the parser with some mechanism for computing the translations.
- To compute the translation at a node A associated with a production $A \rightarrow XYZ$, we need only the values of the translations associated with nodes labeled X, Y and Z.
- These nodes will be roots of subtrees in the forest representing the partially constructed parse tree.
- The nodes X, Y and Z will become children of node A after reduction by $A \rightarrow XYZ$. Once the reduction has occurred we do not need the translations of X, Y and Z any longer.
- One way to implement a syntax-directed translator is to use extra fields in the parser stack entries corresponding to the grammar symbols.
- These extra fields hold the values of the corresponding translations.
- Let us suppose the stack is implemented by a pair of arrays STATE and VAL, as shown in Fig. (a). Each STATE entry is a pointer (or index) to the LR (1) parsing table.
- If the i-th STATE symbol is E, then VAL [i] will hold the value of the translation E. VAL associated with the parse tree node corresponding to this E.
- TOP is a pointer to the current top of the stack. We assume semantic routines are executed just before each reduction.
- Before XYZ is reduced to A, the value of the translation of Z is in VAL [TOP], that of Y in VAL [TOP + 1] and that of X in VAL [TOP + 2].
- After the reduction, TOP is incremented by 2 and the value of A. VAL appears in VAL [TOP].

Example :

- Consider an example of how a syntax-directed translation scheme can be used to specify a "desk calculator" program and how that translation scheme can be implemented by a bottom-up parser that invokes program fragment to compute the semantic actions.
- The desk calculator is to evaluate arithmetic expressions involving integer operands and the operators + and *. We assume that an input expression is terminated by $. The output is to be the numerical value of the input expression. For example, for the input expression 23 * 5 + 4$, the program is to produce the value 119.
- We use the nonterminals S (for complete sentence), E (for expression) and I (for integer). The productions are

$S \rightarrow E\$$

$S \rightarrow E + E$

$S \rightarrow E * E$

$S \rightarrow (E)$

$S \rightarrow I$

S → I digit

I → digit

- The terminals are $, +, *, parentheses and digit, which we assume stands for any of the digits 0, 1, ...9.

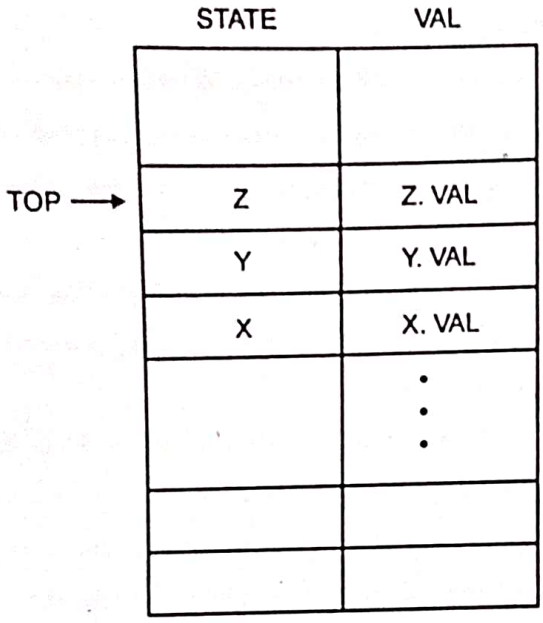


Fig. (a) Stack before reduction

- We must now add the semantic actions to the productions. With each of the nonterminal E and I we associate one integer-valued translation, called E. VAL and I. VAL, respectively, which denotes the numerical value of the expression or integer represented by a node of the parse tree labeled E or I.
- With the terminal digit we associate the translation LEXVAL, which we assume is the second component of the pair (digit, LEXVAL) returned by the lexical analyzer when a token of type digit is found.
- One possible set of semantic actions for the desk calculator grammar is shown in table (1). Using this syntax-directed translation scheme, the input 23 * 5 + 4$ would have the parse tree and translations shown in Fig. (b).
- To implement this syntax-directed translation scheme we need to construct a lexical analyzer and a bottom-up parser and we must make the parser invoke a program fragment to implement a semantic action just before making each reduction.
- A compiler-complier would tie the parser and the semantic action program fragments together, producing one module.

Table (1) Syntax-directed translation scheme for desk calculator

Sr. No.	Production	Semantic action
(1)	$S \rightarrow E\$$	{print E. VAL}
(2)	$E \rightarrow E^{(1)} + E^{(2)}$	{E. VAL : = E. (1) VAL + E. (2) VAL }.
(3)	$E \rightarrow E^{(1)} * E^{(2)}$	{E. VAL : = E. (1) VAL * E. (2) VAL }.
(4)	$E \rightarrow (E^{(1)})$	{E. VAL : = E. (1) VAL}.
(5)	$E \rightarrow I$	{E. VAL : = I.VAL}.
(6)	$I \rightarrow I^{(1)}$ digit	{E. VAL : = 10 * I. (1) VAL + LEXVAL}.
(7)	I → digit	{I. VAL : = LEXVAL}.

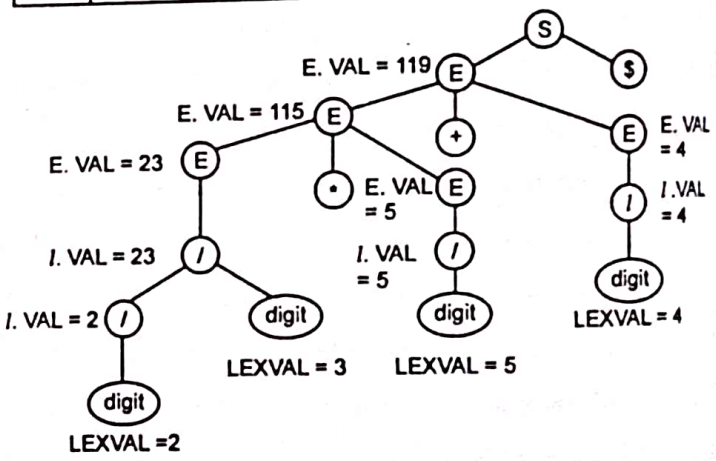


Fig. (b) Parse tree with translation

- The construction of the lexical analyzer is easy. We assume the lexical analyzer scans the input and partitions it into tokens, which here are just the terminals of the grammar.
- Every time the parser calls for a shift and needs the next token, the lexical analyzer skips over blanks to find the next nonblank input symbol, which will be one of the terminals.
- In the case that one of 0, 1, ... 9 is the next nonblank, the lexical analyzer returns the token digit and a value, denoted LEXVAL, which will be the numerical value of the digit found.
- To implement the semantic actions we cause the parser to execute the program fragments of table (2) just before making the appropriate reduction.

Table (2) Implementation of desk calculator

Sr. No.	Production	Semantic action
(1)	S → E$	print VAL [TOP]
(2)	E → E + E	VAL [TOP] : = VAL [TOP] + VAL [TOP – 2]
(3)	E → E * E	VAL [TOP] : = VAL [TOP] * VAL [TOP – 2]
(4)	E → (E)	VAL [TOP] : = VAL [TOP – 1]
(5)	E → I	none
(6)	I → I digit	VAL [TOP] : = 10 VAL [TOP] * VAL [TOP] + LEXVAL
(7)	I → digit	VAL [TOP] : = LEXVAL

- Fig. e shows the sequence of moves made by the parser on input 23 * 5 + 4$.
- The contents of the STATE and VAL fields of the parsing stack are shown after each move.
- We have again taken the liberty of replacing stack states by their corresponding grammar symbols.
- We take the further liberty of using, instead of digit on the stack, its associated LEXVAL.

Table (3) Sequence of moves

Sr. No.	Input	STATE	VAL	Production used
(1)	23 * 5 + 4$	-	-	
(2)	3 * 5 + 4$	2	-	
(3)	3 * 5 + 4$	I	2	I → digit
(4)	* 5 + 4$	I3	2_	
(5)	* 5 + 4$	I	(23)	I → I digit
(6)	* 5 + 4$	E	(23)	E → I
(7)	5 + 4$	E*	(23)_	
(8)	+ 4$	E * 5	(23)_ _	
(9)	+ 4$	E * I	(23)_ 5	I → digit
(10)	+ 4$	E * E	(23)_ 5	E → I
(11)	+ 4$	E	(115)	E → E * E
(12)	4$	E +	(115)_	
(13)	$	E + 4	(115)_ _	
(14)	$	E + I	(115)_ 4	I → digit
(15)	$	E + E	(115)_ 4	E → I
(16)	$	E	(119)	E → E + E
(17)	-	E$	(119)_	
(18)	-	S	-	S → E$

- Consider the sequence of events on seeing the input symbol 2. In the first move the parser shifts the state corresponding to the token digit (Whose LEXVAL is 2) onto the stack (The state is represented by LEXVAL which is 2).
- On the second move the parser reduces by the production I → digit and then invokes the semantic action I. VAL = LEXVAL. The program fragment implementing this semantic action causes the VAL of the stack entry for digit to acquire the value 2.
- Note that after each reduction and semantic action the top of the VAL stack contains the value of translation associated with the left side of the reducing production.

EVALUATION OF EXPRESSIONS USING SEMANTIC ACTIONS

Q.15. Write in brief about evaluation of expression using semantic actions.

Ans. Semantic actions :

- A syntax-directed translation scheme is merely a context-free grammar in which a program fragment called an output action (or sometimes a semantic action or semantic rule) is associated with production.
- For example, suppose output action α is associated with production $A \rightarrow XYZ$. The action α is executed whenever the syntax analyzer recognizes in its input a substring w which has a derivation of the form $A \Rightarrow XYZ \overset{*}{\Rightarrow} w$.
- In a bottom-up parser, the action is taken when XYZ is reduced to A.

- In a top-down parser the action is taken when A, X, Y or Z is expanded, whichever is appropriate.
- The output action may involve the computation of values for variables belonging to the compiler, the generation of intermediate code, the printing of an error diagnostic, or the placement of some value in a table.
- For example, the values computed by action α quite frequently are associated with the parse tree nodes corresponding to the instance of A to which α is reduced.
- A value associated with a grammar symbol is called a translation of that symbol. The translation may be a structure consisting of fields of various types. The rules for computing the value of a translation can be as involved as we wish.
- We shall usually denote the translation fields of a grammar symbol X with names such as X. VAL, X. TRUE, and so forth.
- If we have a production with several instances of the same symbol on the right, we shall distinguish the symbols with superscripts. For example, suppose we have the production and semantic action

$E \rightarrow E^{(1)} + E^{(2)}$

{E. VAL : = $E^{(1)}$. VAL + $E^{(2)}$. VAL}

- The semantic action is enclosed in braces and it appears after the production.
- Here the semantic action is a formula which states that the translation E.VAL associated with the E on the left side of the production is determined by adding together the translations associated with the E's on the right side of the production.
- Note that the terminal symbol + in the productions is "translated" into its usual meaning by the semantic rule.
- This translation is suitable not for a compiler, but for a "desk calculator" program that actually evaluates expressions rather than generating code for them.
- In most compilers we need an action that generates code to perform the addition.
- It defines the value of the translation of the nonterminal on the left side of the production as a function of the translations of the nonterminals in the right side.
- Such a translation is called a synthesized translation.

Consider the following production and action

$A \rightarrow XYZ$ {Y.VAL : = 2 * A.VAL}

- Here the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left. Such a translation is called an inherited translation.
- Consider the following syntax-directed translation scheme suitable for a "desk calculator" program, in which E.VAL is an integer-valued translation.

Production	Semantic action
$S \rightarrow E^{(1)} + E^{(2)}$	{E. VAL : = E. (1) VAL + E. (2) VAL }.
E → digit	{E. VAL : = digit }.

- Here digit stands for any digit between 0 and 9.
- The values of the translations are determined by constructing a parse tree for an input string and then computing the values the translations have at each node.
- For example, suppose we have the input string 1 + 2 + 3. A parse tree for this string is shown in Fig. (a).

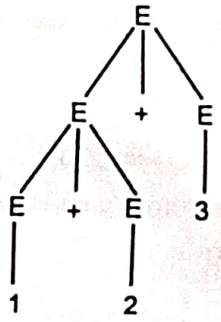


Fig.(a) Parse tree for expression 1 + 2 + 3.

- Consider the bottom leftmost E. This node corresponds to a use of the production $E \rightarrow I$.
- The corresponding semantic actions sets E.VAL = I.
- Thus we can associate the value 1 with the translation E.VAL at the bottom leftmost E.
- Similarly, we can associate the value 2 with the translation E.VAL at the right sibling of this node.

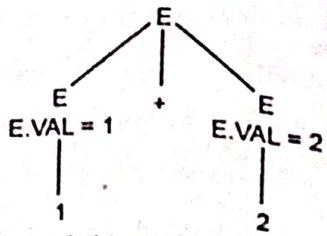


Fig.(b) Subtree with previously computed translations.

- Now consider the subtree shown in Fig. (b). The value of E.VAL at the root of this subtree is 3, which we calculate using the semantic rule

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

- In applying this rule we substitute the value of E.VAL of the bottom left most E for $E^{(1)}.VAL$ and the value of E.VAL at its right sibling E for $E^{(2)}.VAL$.
- Continuing in this manner we derive the values shown in Fig. (c) for the translations at each node of the complete parse tree.

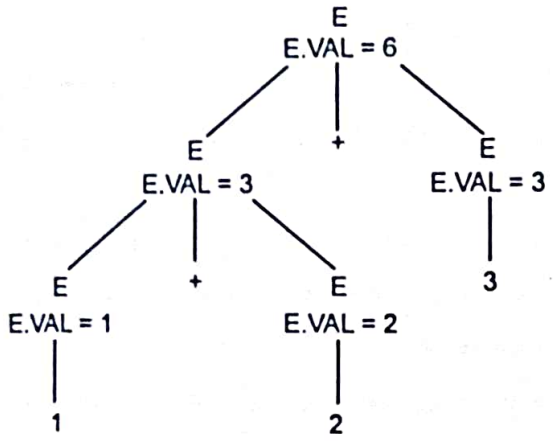


Fig. (c) Complete parse tree

- We see from this example that whenever we apply the rule

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

the values of $E^{(1)}.VAL$ and $E^{(2)}.VAL$ have been computed by a previous application of this rule or the rule E.VAL : = digit. That is to say, if we have a translation A.VAL, then the formulas for all productions with an A on the left side must produce a value that can be used for every occurrence of A. VAL in a semantic rule associated with a production having A on the right.

INTERMEDIATE CODE REPRESENTATION (POSTFIX, SYNTAX TREE, TAC)

Q.16. What are different intermediate forms of source program used in compilers? Discuss the relative advantages and disadvantages.

OR Write short note on intermediate code.

Ans. Intermediate code :

- Many compilers translate the source code into a language which is intermediate in complexity between programming language (high level) and machine code which is called as intermediate code.
- The reason for this is, if source is directly translated into machine code or assembly code, generation of optimal or relatively good code is difficult task.
- Four kinds of intermediate code are often used in compilers, they are:

(1) Postfix notation

(2) Syntax trees

(3) Quadruples

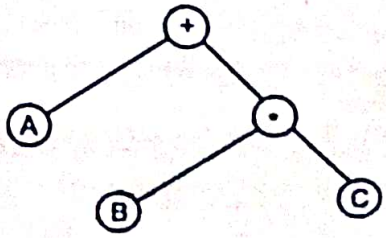
(4) Triples

(1) Postfix notation :

- In this notation, the operator is placed at the right end in the expression example ab+. If e_1 and e_2 are any two postfix expressions and O is a binary operator, the result of applying O to the values denoted by e_1 and e_2 is indicated in the postfix notation by $e_1\ e_2$.
- No parentheses are needed, because the position and priority of the operations permit only one way to decode a postfix expression.
- If O is k operators and $e_1, e_2, e_3, \ldots\ldots e_k$ are postfix expressions, then the results of applying O to these expressions, is denoted as $e_1e_2e_3\ldots e_k$.

(2) Syntax trees :

- This is a representation of source program, especially in optimizing compilers when the intermediate code needs to be extensively restructured.
- A syntax tree is variant of parse tree in which each leaf represents an operand and each interior node an operator, e.g., a syntax tree A+B*C is shown below.



(3) Three-Address Code (Quadruples and Triples) :

- This intermediate code is preferred in many compilers, especially those doing extensive code optimization, because it allows convenient rearrangement of code.
- It consists of sequence of statements, typically of the general form A: = B Op C, where A, B, C are either programmer defined names, constants or compiler generated temporary names; Op stands for operator.
- Since each statement usually contains three addresses two for operands and one for results, such is called as three address code.
- In actual compiler, these statements can be implemented in one of the following ways :

(i) Quadruple :

- It is a list of records with each record consisting of four fields; this representation is known as quadruples. The fields can be labelled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The contents of the rest of the fields are normally pointers to the symbol table entries of names.
- A typical quadruple representation is shown below :

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				
(2)				

(ii) Triples :

- This is a list of records, each consisting of only three fields, which is labelled as OP, ARG1, ARG2 respectively.
- The contents to the records of list itself, because RESULT field is absent.
- Typical triple representation is shown below :

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

(iii) Indirect triple :

In this, a list pointer to the three address statement represented as triple maintained, as shown below.

	LIST	OP	ARG1	ARG2
(0)	(14)	(14) +	A	B
(1)	(15)	(15) *	(14)	C

Comparison :

- The postfix notation is useful if the language is mostly expressive.
- The problem with this notation is in handling flow control.
- One of the solutions is to introduce labels and conditional and unconditional jumps into postfix code.
- The postfix code can then stored in single dimensional array (in which each being either opened).
- Operands are represented by pointer to symbol table and operators by integer codes. Therefore, the table will be just an index of the array.
- Using quadruple, the location for each temporary can be immediately accessed via symbol table. In triple we have no idea unless we scan code.
- In optimizing compiler quadruple permits convenient movement of statement around, whereas in Triples the movement of a statement that defines temporarily value requires us to change all pointer to that statement in ARG1 and ARG2 fields.
- Indirect triples present no such problems because a separate list of pointer into the triple structure is maintained, hence to move the statements recording of this list is required to be done, and no change in the triple structure.
- Hence the utility of the indirect triples is almost same that of quadruples.

Q.17. Explain the term 'Three-address code'.

Ans. Three-address code :

- Syntax trees and postfix notations are of two kinds of intermediate representations.
- 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 complier-generated temporaries; op stands for an operator, such as fixed or floating-point arithmetic operator, or a logical operator on Boolean valued data.

(3) Three-Address Code (Quadruples and Triples) :

- This intermediate code is preferred in many compilers, especially those doing extensive code optimization, because it allows convenient rearrangement of code.
- It consists of sequence of statements, typically of the general form A = B Op C, where A, B, C are either programmer defined names, constants or compiler generated temporary names. Op stands for operator.
- Since each statement usually contains three addresses two for operands and one for results, such is called as three address code.
- In actual compiler, these statements can be implemented in one of the following ways :

(i) Quadruple :

- It is a list of records with each record consisting of four fields; this representation is known as quadruples. The fields can be labelled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The contents of the rest of the fields are normally pointers to the symbol table entries of names.
- A typical quadruple representation is shown below :

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				
(2)				

(ii) Triples :

- This is a list of records, each consisting of only three fields, which is labelled as OP, ARG1, ARG2 respectively.
- The contents to the records of list itself, because RESULT field is absent.
- Typical triple representation is shown below :

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

(iii) Indirect triple :

In this, a list pointer to the three address statement represented as triple maintained, as shown below.

	LIST	OP	ARG1	ARG2
(0)	(14)	(14) +	A	B
(1)	(15)	(15) *	(14)	C

Comparison :

- The postfix notation is useful if the language is mostly expressive.
- The problem with this notation is in handling flow control.
- One of the solutions is to introduce labels and conditional and unconditional jumps into postfix code.
- The postfix code can then stored in single dimensional array (in which each being either opened).
- Operands are represented by pointer to symbol table and operators by integer codes. Therefore, the table will be just an index of the array.
- Using quadruple, the location for each temporary can be immediately accessed via symbol table. In triple we have no idea unless we scan code.
- In optimizing compiler quadruple permits convenient movement of statement around, whereas in Triples the movement of a statement that defines temporarily value requires us to change all pointer to that statement in ARG1 and ARG2 fields.
- Indirect triples present no such problems because a separate list of pointer into the triple structure is maintained, hence to move the statements recording of this list is required to be done, and no change in the triple structure.
- Hence the utility of the indirect triples is almost same that of quadruples.

Q.17. Explain the term 'Three-address code'.

Ans. Three-address code :

- Syntax trees and postfix notations are of two kinds of intermediate representations.
- 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 an operator, such as fixed or floating point arithmetic operator, or a logical operator on Boolean valued data.

- Note that there should be only one operator on the right side of a statement.
- Thus a same language expression like x + y + z might be treated into a sequence

$t_1 := y * z$

$t_2 := x + t_1$

- 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.

Q.18. Explain the data structures used for representation of three address code and compare them. **CS : W-14(6M)**

Ans. Three address codes can be represented by using one of the following data structures :

(1) Quadruple :

- This is a list of record, with each record consisting of four fields. The fields can be labeled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The representation is shown below.

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				

(2) Triples :

- This is a list of records, each consisting of only three fields which can be tabled as OP, ARG1, ARG2 respectively.
- The contents of ARG1 and ARG2 fields are either pointer to the symbol table entries or pointer to the records of list itself, because RESULT fields are absent.
- Typical triple representation is shown below.

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

(3) Indirect triple :

In this, a list of pointer three address statements represented as triples is maintained, as shown below.

	LIST		OP	ARG1	ARG2
(0)	(14)	(14)	+	A	B
(1)	(15)	(15)	*	(14)	C

Comparison :

- Using quadruple, the location for each temporary can be immediates accessed via symbol table.
- In triple we have no idea unless we scan code in optimizing compiler.
- Quadruple permits convenient movement of statements around, whereas in Triples. The movement of a statement in ARG1 Indirect Triple and Quadruples require about same amount of space Indirect Triple can save some space, compared to Quadruples if the same temporary value is used more than once.

Q.19. What are the different types of three-address statements?

Ans. The common three-address statements which we have used are as follows :

(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.

(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 y.

(6) Param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of the three-address statements :

param x_1

param x_2

.....

param x_n

call p, n

generated as part of a call of the procedure p ($x_1, x_2, \ldots, x_n$)

(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.

Q.20. Give the scheme to produce three-address code for assignments.

Ans.

PRODUCTION	SEMANTIC RULES
S → id : E	S. code : = E. code ‖ gen (id. Place ': =' E. place)
$E \rightarrow E_1 + E_2$	E. place : = newtemp; E.code : = E_1 .code ‖ E_2 . code ‖ Gen(E.place ': =' E_1 . Place '+' E_2 . Place)
$E \rightarrow E_1 * E_2$	E.place : = newtemp; E.code : = E_1 .code ‖ E_2 . code ‖ Gen(E.place : =' E_1 place '*' E_2 place)
$E \rightarrow -E_1$	E. place : = newtemp; E.code:= E_1 code ‖ Gen (E.place ': =' 'Uminus' E_1 place).
$E \rightarrow (E_1)$	E. place : = E_1 . place; E.code := E_1 . code
E → id	E.place : = id. place; E. code : = "

Now, the scheme to produce three-address code for assignment is

S → id : "E {p : = lookup (id. name);
if p ≠ nil, then
emit (p ': = ' E. place)
else error}

$E \rightarrow E_1 + E_2$ {E. place : = newtemp;
emit (E. place ': = ' E_1 place '+' E_2 . place)}

$E \rightarrow E_1 * E_2$ {E place : = newtemp;
emit (E place ': = ' E_1 . Place ': = ' '*' E_2 . place)}

$E \rightarrow -E_1$ {E.place : = newtemp;
emit (E place ': = ' 'Uminus ' E_1 . place)}

$E \rightarrow (E_1)$ {E. place : = E_1 . place}

E → id {p : = lookup (id.name);
if p ≠ nil, then
E. place : = p
else error}

Q.21. Translate following expression

– (a * b) * (c + d) / (a * b + c) into

(i) Quadruples

(ii) Triples

(iii) Indirect triple CT : S-09(6M)

Ans.

(i) Quadruple :

Sr. No.	Operator	operand 1	operand 2	result
(1)	*	a	b	t_1
(2)	uniminus	t_1		t_2
(3)	+	c	d	t_3
(4)	*	t_2	t_3	t_4
(5)	+	t_1	c	t_5
(6)	/	t_4	t_5	t_6

(ii) Triples :

Sr. No.	Operator	operand 1	operand 2
(1)	*	a	b
(2)	uniminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	+	(4)	(5)

(iii) Indirect triples :

Sr. No.	Statement	Operator	operand 1	operand 2
(1)	(14)	+	a	b
(2)	15	–	(14)	-
(3)	16	+	c	d
(4)	17	*	15	16
(5)	(18)	+	14	c
(6)	19	+	17	18

Translate the following expression :

-(a + b) * (c + d) + (a + b + c) into

(i) Quadruple

(ii) Triples

(iii) Indirect triples.

Describe various TAC and translate the expression

-(a + b) * (c + d) + (a + b + c) into quadruple and triples.

CT : W-11(6M), S-13,W-13(5M)

Quadruple :

Sr. No.	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t2
(2)	-	t1		t2
(3)	+	c	d	t3
(4)	*	t2	t3	t4
(5)	+	t1	c	t5
(6)	+	t4	t5	t6

Triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	-	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

(iii) Indirect triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	–	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

Q.23. Generate TAC for following statement using SDTS.

A : = – B * (C + D). **CS : S-11,12(3M)**

Ans. The three address code will be

$t_1 = - B_1$

$t_2 = C + D$

$t_3 = t_1 * t_2$

$a = t_3$

POINTS TO REMEMBER :

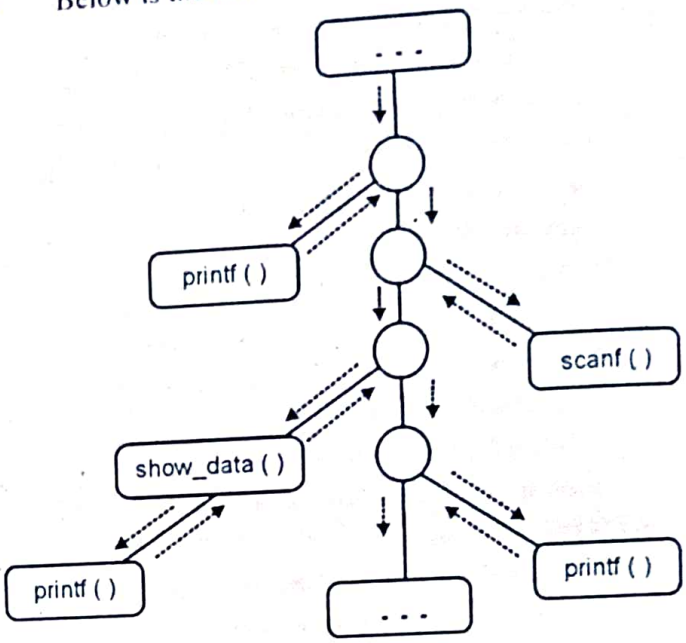
1. *The attributes associated with a grammar symbol are classified into two categories :*
   (i) *Synthesized attributes.*
   (ii) *Inherited attributes.*
2. *An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes.*
3. *Inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attributes of the parsing or siblings of that node.*
4. *While translating a source program into a functionally equivalent object code representation, a parser may first generate an intermediate representation that is called as intermediate code generation.*

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e. parameters which are used to send input to the called procedure.
Return Value	Stores return values.

- Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.
- We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.
- To understand this concept, we take a piece of code as an example:

```
...
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue...");
...
int show_data(char *user)
{
    printf("Your name is %s", username);
    return 0;
}
...
```

- Below is the activation tree of the code given.



- The procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

PARAMETER PASSING, VALUE RETURN

Q.9. Explain Parameter Passing.

Ans. Parameter Passing:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

(i) r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

(ii) **l-value:**

- The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

- From this example, we understand that constant values like 1, 7, 12 and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

(1) **Formal Parameters:**

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

(2) **Actual Parameters:**

- Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
}
fun_two(int formal_parameter)
{
    print formal_parameter;
}
```

- Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

(a) **Pass by Value:**

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

(b) **Pass by Reference:**

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

(c) **Pass by Copy-restore:**

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

(d) Pass by Name:

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

MEMORY ALLOCATION STORAGE ALLOCATION

Q.10. Explain Storage Allocation.

OR Explain Storage Allocation. What are the different Storage Allocations?

Ans. Storage Allocation:

Runtime environment manages runtime memory requirements for the following entities:

(i) Code:

It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

(ii) Procedures:

Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.

(iii) Variables:

Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Types of Allocation:

(1) Static Allocation

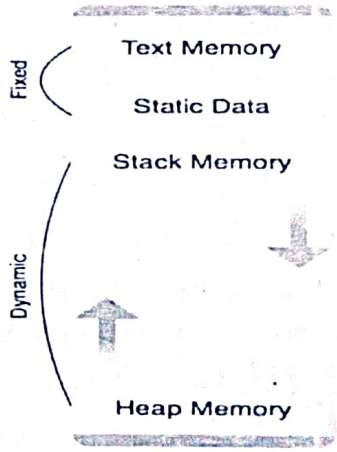
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

(2) Stack Allocation:

Procedure calls and their activations are managed by means of stack memory allocation. It works in Last-In-First-Out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

(3) Heap Allocation

- Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



- As shown in the above Fig. the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Q.11.Explain Memory allocation.

OR What are the techniques of storage allocation in compiler design?

Ans. There are various storage allocation techniques which are as follows:

(1) **Static Allocation:**

- It is the simplest allocation scheme in which allocation of data objects is done at compile time because the size of every data item can be determined by the compiler.
- Recursive Subprogram and Arrays of adjustable length are not permitted in a language. In static allocation, the compiler can decide the amount of storage needed by each data object. Thus, it becomes easy for a compiler to identify the address of these data in the activation record.
- FORTRAN uses this kind of storage allocation strategies.

Advantages:

(i) It is easy to implement.
(ii) It allows type checking during compilation.
(iii) It eliminates the feasibility of running out of memory.

Disadvantages:

(i) It is incompatible with recursive subprograms.
(ii) It is not possible to use variables whose size has to be determined at run time.
(iii) The static allocation can be completed if the size of the data object is called compile time.

(2) **Dynamic Allocation (Stack Allocation):**

- The stack allocation is a runtime storage management technique. The activation records are pushed and popped as activations begin and end respectively.
- Storage for the locals in each call of the 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 the call is made.
- It can be determined the size of the variables at a run time & hence local variables can have different storage locations & different values during various activations.
- It allows recursive subprograms.
- ALGOL language uses this strategy.
- On each execution of a procedure, an Activation Record is generated, which contains information like local data, actual parameter, return value, return address of a procedure. The Activation Record for that procedure is saved onto the stack.

Advantages:

(i) It supports recursion.
(ii) It creates a data structure for the data item dynamically.

Disadvantage:

Memory Addressing can be done using pointers and index Registers.

(3) **Heap Storage Allocation**

- It enables the allocation of memory in a non-nested design. Storage can be allocated and freed arbitrarily from an area known as Heap.
- Heap Allocation is helpful for executing data whose size varies as the program is running.
- Heap is maintained as a list of free space called free space list.

Advantage:

A large block of storage can be partitioned into smaller blocks at run time.

Disadvantage:

It creates the problem of fragmentation.

Q.12. Differentiate between Stack allocation and Heap allocation.

Ans.

Sr. No.	Static allocation	Heap allocation
(1)	Static allocation allocates memory on the basis of size of data objects.	Heap allocation makes use of heap for managing the allocation of memory at run time.
(2)	In static allocation, there is no possibility of creation of dynamic data structures and objects.	In heap allocation, dynamic data structures and objects are created.
(3)	In static allocation, the names of the data objects are fixed with storage for addressing.	Heap allocation allocates contiguous block of memory to data objects.

(4)	Static allocation is simple, but not efficient memory management technique.	Heap allocation does memory management in efficient way.
(5)	Static allocation strategy is faster in accessing data as compared to heap allocation.	Heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space.
(6)	It is easy to implement.	It is comparatively expensive to implement.

UNIT - IV

GLOBAL OPTIMIZATION

Q.22. Explain Global Optimization.

Ans. Global Optimization:

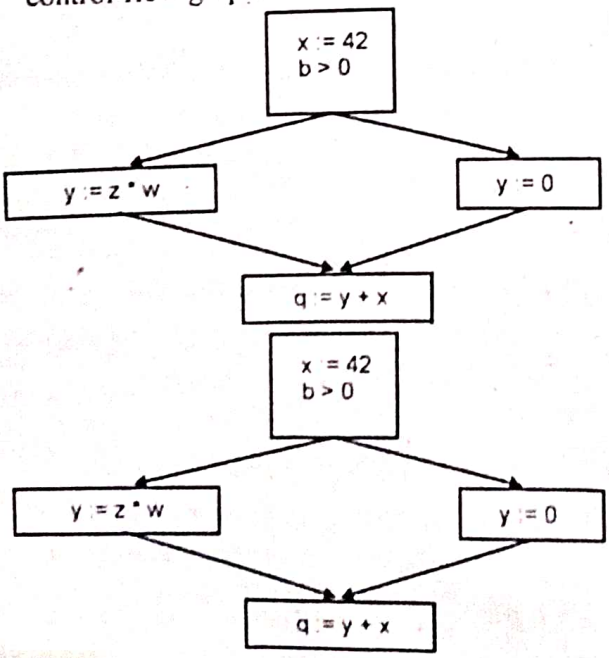
The scope of Global optimization is generally a program unit, usually spanning over number of basic blocks.

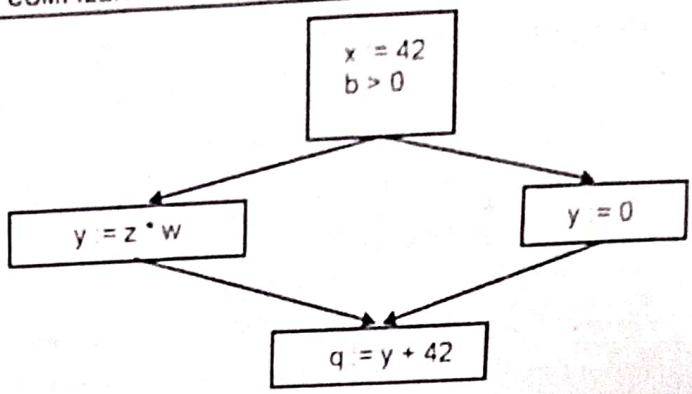
- It can achieved better transformation leading to more optimized.
- Program flow graph normally undergoes two types analysis:

(a) Control Flow Analysis.

(b) Data Flow Analysis.

- These optimizations can be extended to an entire control-flow graph.





UNIT - V

ARCHITECTURE DEPENDENT CODE IMPROVEMENT IN COMPILER DESIGN: INSTRUCTION SCHEDULING

Q.23. Explain Architecture dependent code improvement in compiler design: Instruction Scheduling.

OR Explain Code Optimizing Process.

OR What is Code Optimization?

Ans. Code Optimization:

- Code optimization is a program transformation approach that aims to enhance code by reducing resource consumption (i.e. CPU and memory) while maintaining high performance.
- In code optimization, high-level generic programming structures are substituted with low-level programming codes. The three guidelines for code optimization are as follows:

(i) In no way should the output code alter the program's meaning.

(ii) The program's speed should be increased, and it should use fewer resources if at all feasible.

(iii) The optimization step should be quick and not hinder the compilation process.

- At several stages of the compilation process, efforts to optimize the code might be made. Users can alter/rearrange the code at first or create the code using better algorithms. The compiler can improve loops and address computations after completing intermediate code. The compiler can leverage Memory Hierarchy and CPU registers while generating the target machine code.