CD CAT1 QB – Answers

Q.1 a) What do you mean by SDTS. Explain with example.

Ans Q.1 a)

Syntax Directed Translation Scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Implementation Of Syntax Directed Translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parse the input and produce a parse tree as a result.

Example

Parse tree for SDT:

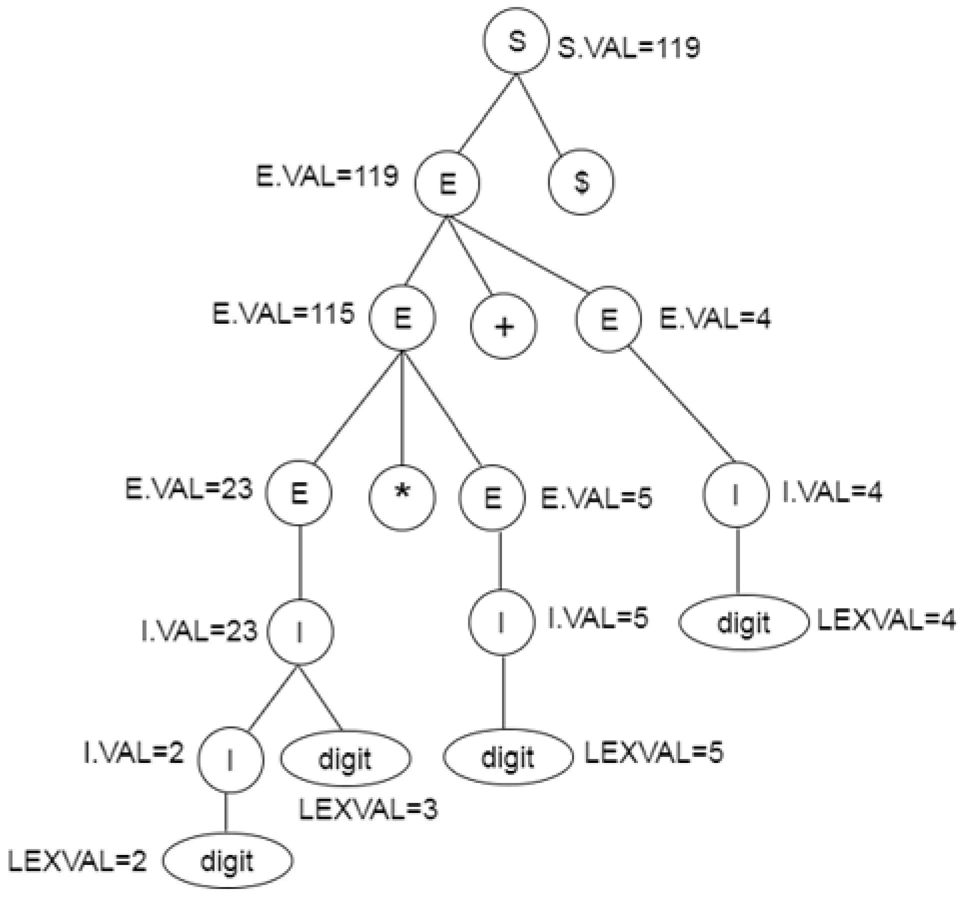


Fig: Parse tree

Q.1 b) Define Attribute. Explain different types of attributes.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

```
S → 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.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

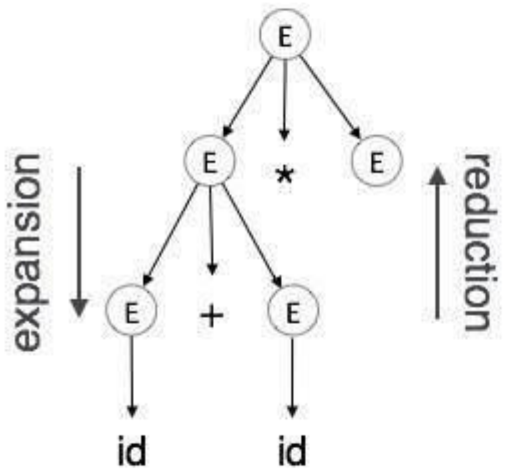
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → 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.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

```
int value  = 5;
<type, "integer">
<presentvalue,  "5">
```

Q.2 a) Translate the expression
A: = -B * (C + D)/E

Q.2 b) Translate given expression into TAC
if x < y then a= b + c else p= q+r

Q.3 a) What is dominator? How it is used to identify loop in three address code?

Dominators are used to identify the leaders of a control flow graph and to optimize loops.

Given two nodes, ‘d’ and ‘n’, we say, node ‘d’ dominates ‘n’, denotes as “d dom n”, if every path from the initial node of the control flow graph (CFG) to ‘n’ goes through ‘d’.

The loop entry dominates all nodes in the loop.
The immediate dominator ‘m’ of a node ‘n’ is the last dominator on the path from the initial node to ‘n. If $d \neq n$ and “d dom n” then “d dom m”.
Dominator Trees are used to indicate which node dominates the other nodes.
Consider figure 35.8, having nearly 10 nodes.

- The root of the CFG is node “1”.
- Thus without reaching 1, we cannot go to any other node.
- Thus 1 dominates all other nodes.
- Consider node 2, which is reached from 1.
- Node 3 is also reached from 1.
- Thus 2 is not a dominator node as this node could be skipped to reach node 3.
- After reaching node 3, this is the key to reach all nodes and there is a simple edge from node 3 to 4
- And thus resulting in node 4 being dominant to all the nodes that are below node 4.

Q.3 b) Write a note on
a) Loop unrolling.
b) Loop Jamming.
c) Loop invariant computation.

a) Loop unrolling.

The loop unrolling technique transforms the loop. In this technique, the number of jumps and tests can be optimized by writing the code to times without changing the meaning of the code. It reduces the number of iterations and increases the program's speed by eliminating the loop control instructions and loop test instructions.

Let's take an example to understand this technique.

Example: Consider the following piece of code.

```
#include <stdio.h>

void main() {
   int i = 1;
   int a[100], b[100];
   while(i<100) {
   a[i] = b[i];
   i++;
   }
}
```

In the above code, a loop is running **100 times**. We can optimize this code by repeating the statements so that the loop runs only **50 times**.

Hence, the optimized code will be-

```
#include <stdio.h>

void main() {
   int i = 1;
   int a[100], b[100];
   while(i<100) {
      a[i] = b[i];
      i++;
      a[i] = b[i];
      i++;
   }
}
```

b) Loop Jamming.

Also known as **Loop Jamming**, this technique combines two or more loops which have the same index variable and number of iterations.
This technique reduces the time taken in compiling all the loops.
Let’s take an example to understand this technique.

Example: Consider the following piece of code.

```
#include <stdio.h>

void main() {
   int a[10], b[10], i;
   for(i = 0; i < 10; i++)
      a[i] = 1;
   for(i = 0; i < 10; i++)
      b[i] = 2;
}
```

In the above code, there are **two loops**. The first loop running **ten times assigns the value 1 at each index of the array a** while the second loop also **runs ten times, assigning the value 2 at each index of the array b**. We can observe that the work of both these loops is almost the same. Both are running ten times and assigning a value to an array. Thus, we can combine both these loops to optimize our code.
Hence, the optimized code will be-

```
#include <stdio.h>

void main() {
   int a[10], b[10], i;
   for(i = 0; i < 10; i++) {
      a[i] = 1;
      b[i] = 2;
   }
}
```

c) Loop invariant computation.

A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop. (Note that this says nothing about its truth or falsity part way through an iteration.)

A loop invariant is some predicate (condition) that holds for every iteration of the loop.

For example, let's look at a simple for loop that looks like this:

```
int j = 9;
for(int i=0; i<10; i++)
j--;
```

In this example it is true (for every iteration) that i + j == 9.

A weaker invariant that is also true is that i >= 0 && i <= 10.

One may get confused between the loop invariant, and the loop conditional (the condition which controls termination of the loop).

The loop invariant must be true:

- before the loop starts
- before each iteration of the loop
- after the loop terminates

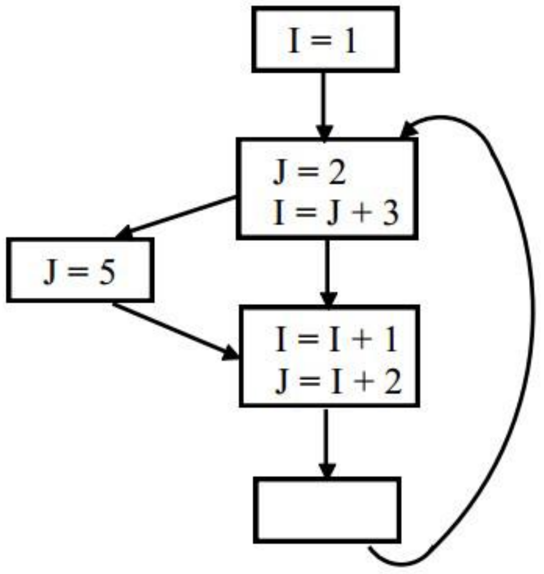
(although it can temporarily be false during the body of the loop).

On the other hand the loop conditional must be false after the loop terminates, otherwise, the loop would never terminate.

A good loop invariant should satisfy three properties:

- **Initialization:** The loop invariant must be true before the first execution of the loop.
- **Maintenance:** If the invariant is true before an iteration of the loop, it should be true also after the iteration.
- **Termination:** When the loop is terminated the invariant should tell us something useful, something that helps us understand the algorithm.

Q.4 a) Find IN and OUT for every blocks for the following graph.



Q.4 b) Explain Peephole optimization with their characteristics.

Peephole optimization is an optimization technique by which code is optimized to improve the machine's performance. More formally, **Peephole optimization is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the peephole optimization in compiler design or window**.

Some important aspects regarding peephole optimization:

1. It is applied to the source code after it has been converted to the target code.
2. Peephole optimization comes under machine-dependent optimization. Machine-dependent optimization occurs after the target code has been generated and transformed to fit the target machine architecture. It makes use of CPU registers and may make use of absolute memory references rather than relative memory references.
3. It is applied to a small piece of code, repeatedly.

The objectives of peephole optimization are as follows:

1. Improve performance
2. Reduce memory footprint
3. Reduce code size.

Peephole Optimization Characteristics

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

Initial code:

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

Optimized code:

```
y = x + 5;
i = y;
w = y * 3;
```

B. Constant folding: The code that can be simplified by the user itself, is simplified.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

```
y = x * 2;
```

Optimized code:

```
y = x + x;    or     y = x << 1;
```

Initial code:

```
y = x / 2;
```

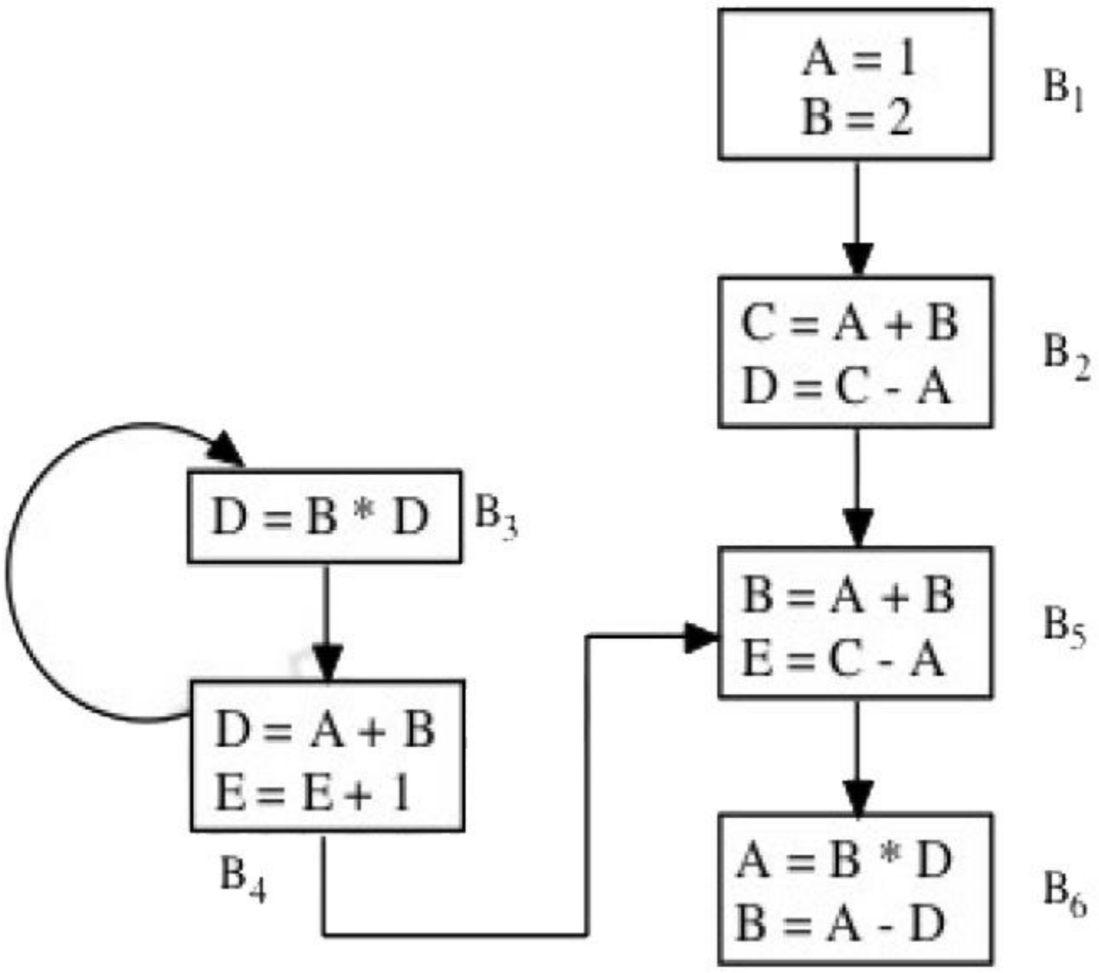
Optimized code:

```
y = x >> 1;
```

D. Null sequences: Useless operations are deleted.

E. Combine operations: Several operations are replaced by a single equivalent operation.

Q.5 a) What is data flow equations? Solve the data flow equation for the following flow graph.



It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis, optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information that can be used for optimization.

Basic Terminologies –

- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.

Q.5b) Use simple code generation algorithm to generate code for following three address codes. Assume two registers are available.
T1 = a + b
T2 = c + d
T3 = e – T2
T4 = T1 – T3

Q.6 a) What are the different issues in code generator?

The Different Issues in Code Generator

Main issues in the design of a code generator are:

- Input to the code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Evaluation order

1. Input to the code generator

In the input to the code generator, design issues in the code generator intermediate code created by the frontend and information from the symbol table that defines the run-time addresses of the data objects signified by the names in the intermediate representation are fed into the code generator. Intermediate codes may be represented mainly in quadruples, triples, indirect triples, postfix notation, syntax trees, DAGs(Directed Acyclic Graph), etc. The code generation step assumes that the input is free of all syntactic and state semantic mistakes, that all essential type checking has been performed, and that type-conversion operators have been introduced where needed.
Also See, Top Down Parsing

2. Target program

The code generator's output is the target program. The result could be:

- **Assembly language:** It allows subprograms to be separately compiled.
- **Relocatable machine language:** It simplifies the code generating process.
- **Absolute machine language:** It can be stored in a specific position in memory and run immediately.

3. Memory management

In the memory management design, the source program's frontend and code generator map names address data items in run-time memory. It utilizes a symbol table. In a three-address statement, a name refers to the name's symbol-table entry. Labels in three-address statements must be transformed into instruction addresses.

For example,
j: goto i generates the following jump instruction:
if i < j, A backward jump instruction is generated with a target address equal to the quadruple i code location.
If i > j, It's a forward jump.
The position of the first quadruple j machine instruction must be saved on a list for quadruple i.
When i is processed, the machine locations for all instructions that forward hop to i are populated.

4. Instruction selection
In the Instruction selection, the design issues in the code generator program's efficiency will be improved by selecting the optimum instructions. It contains all of the instructions, which should be thorough and consistent. Regarding efficiency, instruction speeds and machine idioms have a big effect.
Instruction selection is simple if we don't care about the target program's efficiency.
The relevant three-address statements, for example, would be translated into the following code sequence:

```
P:=Q+R
S:=P+T
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

The fourth sentence is unnecessary since the P value is loaded again in that statement already stored. It results in an inefficient code sequence. A given intermediate representation can be translated into several distinct code sequences, each with considerable cost differences. Previous knowledge of instruction cost is required to build good sequences, yet reliable cost information is difficult to forecast.

5. Register allocation
In the Register allocation, design issues in the code generator can be accessed faster than memory. The instructions involving operands in the register are shorter and faster than those involved in memory operands.
The following sub-problems arise when we use registers:

- **Register allocation:** In register allocation, we select the set of variables that will reside in the register.
- **Register assignment:** In the Register assignment, we pick the register that contains a variable.

Certain machines require even-odd pairs of registers for some operands and results.

Example

Consider the following division instruction of the form:

D **x**, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

An old register is used to hold the quotient.

6. Evaluation order

The code generator determines the order in which the instructions are executed. The target code's efficiency is influenced by order of computations. Many computational orders will only require a few registers to store interim results. However, choosing the best order is a completely challenging task in the general case.

Q. 6 b) Explain Register allocation and assignment.

Register allocation and assignment

Allocation –

Maps an unlimited namespace onto that register set of the target machine.

- **Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.
- **Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models the physical register set.

Allocation ensures that code will fit the target machine's reg. set at each instruction.

Assignment –

Maps an allocated name set to the physical register set of the target machine.

- Assumes allocation has been done so that code will fit into the set of physical registers.
- No more than **'k'** values are designated into the registers, where 'k' is the no. of physical registers.

Local Register Allocation and Assignment:
Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.
Top-Down Approach is a simple approach based on 'Frequency Count'. Identify the values which should be kept in registers and which should be kept in memory.

Algorithm:

1. Compute a priority for each virtual register.
2. Sort the registers into priority order.
3. Assign registers in priority order.
4. Rewrite the code.

Global Register Allocation and Assignment:
1. The main issue of a register allocator is minimizing the impact of spill code;

- Execution time for spill code.
- Code space for spill operation.
- Data space for spilled values.

2. Global allocation can't guarantee an optimal solution for the execution time of spill code.
3. Prime differences between Local and Global Allocation:

- The structure of a global live range is naturally more complex than the local one.
- Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)

4. To make the decision about allocation and assignments, the global allocator mostly uses graph coloring by building an interference graph.
5. Register allocator then attempts to construct a k-coloring for that graph where 'k' is the no. of physical registers.

- In case, the compiler can't directly construct a k-coloring for that graph, it modifies the underlying code by spilling some values to memory and tries again.
- Spilling actually simplifies that graph which ensures that the algorithm will halt.

6. Global Allocator uses several approaches, however, we'll see top-down and bottom-up allocations strategies. Subproblems associated with the above approaches.

- Discovering Global live ranges.
- Estimating Spilling Costs.
- Building an Interference graph

Q.7 a) Define a DAG. Draw the DAG for the following three address code.
d = b * c
e = a + b
b = b * c
a = e - d

Q7 b) What are different storage allocation strategies? Explain.

The different strategies to allocate memory are:

1. Static storage allocation
2. Stack storage allocation
3. Heap storage allocation

I. **Static storage allocation**
   - In static allocation, names are bound to storage locations.
   - If memory is created at compile time then the memory will be created in static area and only once.
   - Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
   - The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
   - Another drawback is restriction of the recursion procedure.

II. **Stack Storage Allocation**
   - In static storage allocation, storage is organized as a stack.
   - An activation record is pushed into the stack when activation begins and it is popped when the activation end.
   - Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
   - It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

III. **Heap Storage Allocation**

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

Example:

```
fact (int n)
{
if (n<=1)
return 1;
else
return (n * fact(n-1));
}
fact (6)
```

Q.8 a) Explain error recovery in lexical analysis phase.

Lexical Errors

A character sequence which is not possible to scan into any valid token is a lexical error.

Important facts about the lexical error:

- Lexical errors are not very common, but it should be managed by a scanner
- Misspelling of identifiers, operators, keyword are considered as lexical errors
- Generally, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery in Lexical Analyzer

Here, are a few most common error recovery techniques:

- Removes one character from the remaining input
- In the panic mode, the successive characters are always ignored until we reach a well-formed token
- By inserting the missing character into the remaining input
- Replace a character with another character
- Transpose two serial characters

Q.8 b) Explain error Recovery in LR parsing.

Error Recovery in LR Parsing:

When there is no valid continuation for the input scanned thus far, LR parsers report an error. Before notifying a mistake, a CLR parser never performs a single reduction and an SLR or LALR may do multiple reductions, but they will never move an incorrect input symbol into the stack.

When the parser checks the table and discovers that the relevant action item is empty, an error is recognized in LR parsing. Goto entries can never be used to detect errors.

LR Parser Basically Uses the Mentioned Two Techniques to Detect Errors:

1. Syntactic Phase recovery
2. Panic mode recovery

I. Syntactic Phase Recovery:

Syntactic Phase Recovery Follows the Given Steps:

1. Programmer mistakes that call error procedures in the parser table are determined based on the language.
2. Creating error procedures that can alter the top of the stack and/or certain symbols on input in a way that is acceptable for table error entries.

There are some of the errors that are detected during the syntactic phase recovery:

- Errors in structure
- Missing operator
- Misspelled keywords
- Unbalanced parenthesis

II. Panic Mode Recovery:

This approach involves removing consecutive characters from the input one by one until a set of synchronized tokens is obtained. Delimiters such as or are synchronizing tokens. The benefit is that it is simple to implement and ensures that you do not end up in an infinite loop. The drawback is that a significant quantity of data is skipped without being checked for additional problems.

Panic mode recovery follows the given steps:

1. Scan the stack until you find a state ‘a’ with a goto() on a certain non-terminal ‘B’ (by removing states from the stack).
2. Until a symbol ‘b’ that can follow ‘B’ is identified, zero or more input symbols are rejected.

Q.9 a) Explain data structures for symbol table organization.

1. **Linked List –**
   - This implementation is using a linked list. A link field is added to each record.
   - Searching of names is done in order pointed by the link of the link field.
   - A pointer **"First"** is maintained to point to the first record of the symbol table.
   - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
2. **Hash Table –**
   - In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
   - A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
   - To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
   - Insertion and lookup can be made very fast – O(1).
   - The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.
3. **Binary Search Tree –**
   - Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
   - All names are created as child of the root node that always follows the property of the binary search tree.
   - Insertion and lookup are O($\log_2$ n) on average.

Q.9 b) Explain run time storage allocation for procedure call and return statement.

1. Implementation of Call statement:

```
1. ADD #caller.recordsize, SP/* increment stack pointer */
2. MOV #here + 16, *SP          /*Save return address */
3. GOTO callee.code_area
```

Where,

caller.recordsize is the size of the activation record

#here + 16 is the address of the instruction following the **GOTO**

2. Implementation of Return statement:

```
1. GOTO *0 ( SP ) /*return to the caller */
2. SUB #caller.recordsize, SP /*decrement SP and restore to previous value */
```

Q.10 a) What are problems in code generation?

Problems in code generation are as follows:-

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to the code generator

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a)Postfix notation
  b)Syntax tree
  c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.10s

c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

```
1. MOV b, R0            R0→b
2. ADD c, R0   R0      c + R0
3. MOV R0, a            a  →  R0
4. MOV a, R0      R0→ a
5. ADD e, R0            R0  →     e + R0
6. MOV R0, d            d   →  R0
```

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

- **Register allocation:** In register allocation, we select the set of variables that will reside in register.
- **Register assignment:** In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

For example:

Consider the following division instruction of the form:

1. D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

Q.10 b) Define symbol table. Explain data structure use for Representation of symbol table

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the 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 of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.
- It is used by various phases of the compiler as follows:-
  1. **Lexical Analysis:** Creates new table entries in the table, for example like entries about tokens.
  2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
  3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e., to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
  4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
  5. **Code Optimization:** Uses information present in the symbol table for machine-dependent optimization.
  6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Data Structure use for Representation of symbol table

1. **<u>Linked List</u> –**
   - This implementation is using a linked list. A link field is added to each record.
   - Searching of names is done in order pointed by the link of the link field.
   - A pointer **“First”** is maintained to point to the first record of the symbol table.

- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

2. **Hash Table** –
   - In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
   - A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
   - To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
   - Insertion and lookup can be made very fast – O(1).
   - The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

3. **Binary Search Tree** –
   - Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
   - All names are created as child of the root node that always follows the property of the binary search tree.
   - Insertion and lookup are O($\log_2$ n) on average.

Q.11 a) What are the different categories and goals of Error handling.

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error. During this whole process processing time of the program should not be slow.
Functions of Error Handler:

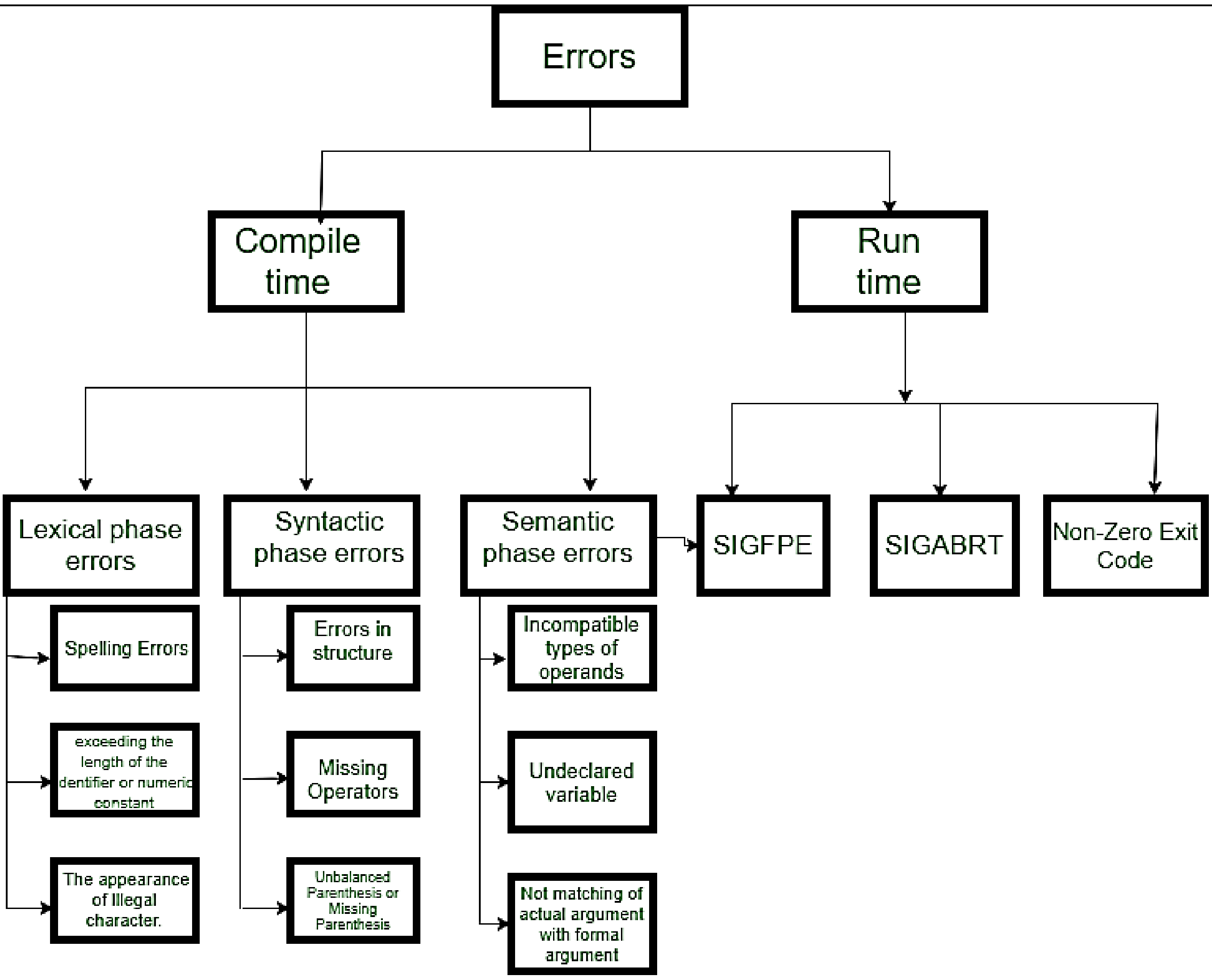
- **Error Detection**
- **Error Report**
- **Error Recovery**

Error handler=Error Detection+Error Report+Error Recovery.

An **Error** is the blank entries in the symbol table.
Errors in the program should be detected and reported by the parser. Whenever an error occurs, the parser can handle it and continue to parse the rest of the input. Although the parser is mostly responsible for checking for errors, errors may occur at various stages of the compilation process.

So, there are many types of errors and some of these are:
Types or **Sources of Error –**
There are three types of error:
logic, run-time and compile-time error:

1. **Logic errors** occur when programs operate incorrectly but do not terminate abnormally (or crash). Unexpected or undesired outputs or other behaviour may result from a logic error, even if it is not immediately recognized as such.
2. A **run-time error** is an error that takes place during the execution of a program and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error is an example of this. Logic errors occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
3. **Compile-time errors** rise at compile-time, before the execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is an example of this.



Errors
Compile time
Run time
Lexical phase errors
Syntactic phase errors
Semantic phase errors
SIGFPE
SIGABRT
Non-Zero Exit Code
Spelling Errors
exceeding the length of the identifier or numeric constant
The appearance of Illegal character.
Errors in structure
Missing Operators
Unbalanced Parenthesis or Missing Parenthesis
Incompatible types of operands
Undeclared variable
Not matching of actual argument with formal argument

Classification of Compile-time error –

1. **Lexical** : This includes misspellings of identifiers, keywords or operators
2. **Syntactical** : a missing semicolon or unbalanced parenthesis
3. **Semantical** : incompatible value assignment or type mismatches between operator and operand
4. **Logical** : code not reachable, infinite loop.

Finding error or reporting an error – Viable-prefix is the property of a parser that allows early detection of syntax errors.

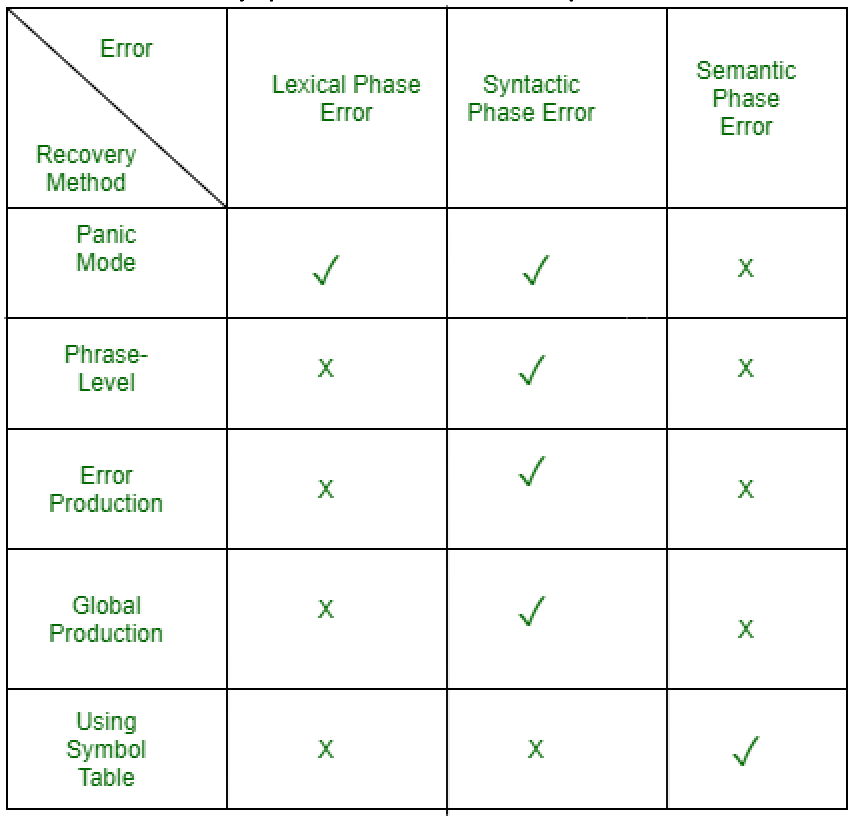
- **Goal** detection of an error as soon as possible without further consuming unnecessary input
- **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

Example: for(**;**), this will report an error as for having two semicolons inside braces.

Error Recovery –

The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are as follows.

We already discuss the errors. Now, let's try to understand the recovery of errors in every phase of the compiler.



Error Recovery Method	Lexical Phase Error	Syntactic Phase Error	Semantic Phase Error
Panic Mode	✓	✓	x
Phrase-Level	x	✓	x
Error Production	x	✓	x
Global Production	x	✓	x
Using Symbol Table	x	x	✓

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

Q.11 b) Write short notes on Register allocation and assignment.

Local register allocation

- ✓ Register allocation is only within a basic block.
- ✓ It follows top-down approach.
- ✓ Assign registers to the most heavily used variables
  - Traverse the block
  - Count uses
  - Use count as a priority function
  - Assign registers to higher priority variables first

Advantage

- Heavily used values reside in registers

Disadvantage

- Does not consider non-uniform distribution of uses

Need of global register allocation

Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.

To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Register allocation by graph coloring

Global register allocation can be seen as a graph coloring problem.
Basic idea:

- Identify the live range of each variable
- Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
- Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

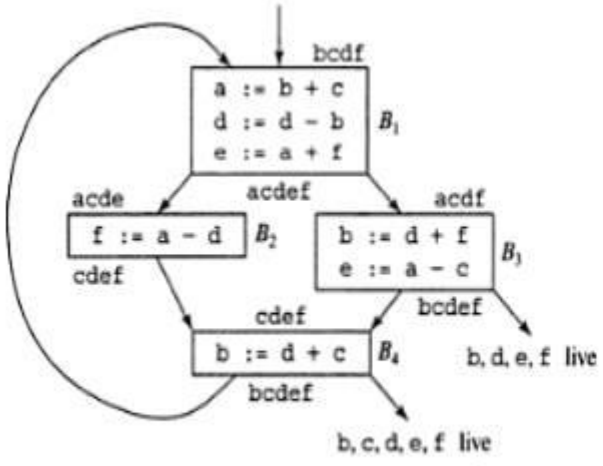


Fig 4.3 Flow graph of an inner loop

Fig 4.3 Flow graph of an inner loop

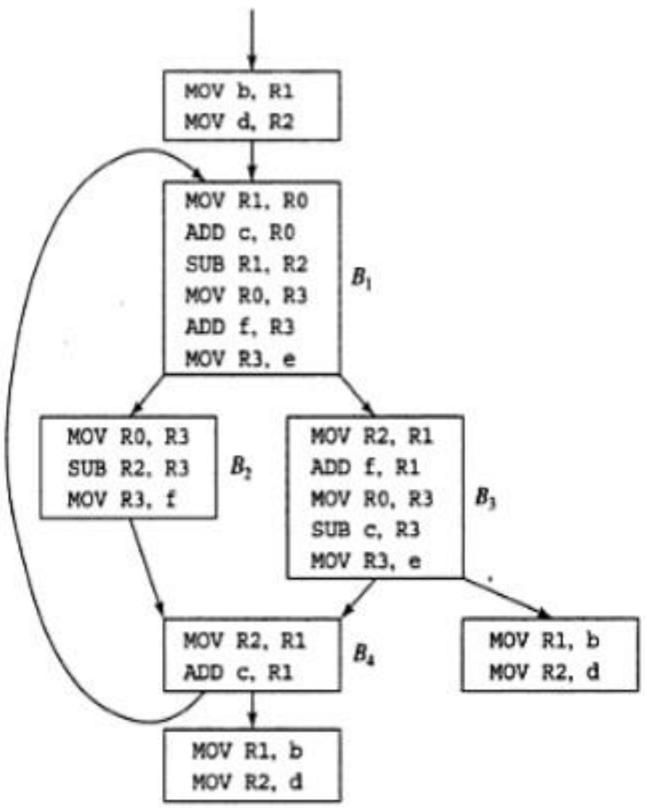


Fig 4.4 Code sequence using global register assignment

Fig 4.4 Code sequence using global register assignment

Q.12 a) Explain the symbol table management for block structured language.

Organization for Block Structured Languages:

- The block structured language is a kind of language in which sections of source code is within some matching pair of delimiters such as “{“ and “}” or begin and end
- Such a section gets executed as one unit or one procedure or a function or it may be controlled by some conditional statements (if, while, do-while)
- Normally, block structured languages support structured programming approach

  Example: C, C++, JAVA, PASCAL, FORTRAM, LISP and SNOBOL
- Non-block structured languages are LISP, FORTRAN and SNOBOL

Implementation of Symbol Table:

- Each entry in the symbol table can be implemented as a record with several fields
- The following data structures are used for organization of block structured languages:

  Linear List
  Self-Organizing List
  Hashing
  Tree Structure

Symbol table organization using Linear List:

- A linear list of records is the easiest way to implement the symbol table
- In this method, an array is used to store names and associated information
- The new names are added to the symbol table in the order they arrive
- The pointer “available” is maintained at the end of all stored records
- To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name we get an error “use of undeclared name”
- While inserting a new name we should ensure that it is not already present. If it is already present then another error occurs, i.e., “Multiple Defined Name”.

Q.12 b) What are the various attribute that should be stored in symbol table and discuss various data structure for implementation of symbol table.

Symbol Table entries – Each entry in the symbol table is associated with attributes that support the compiler in different phases.

Use of Symbol Table-

The symbol tables are typically used in compilers. Basically compiler is a program which scans the application program (for instance: your C program) and produces machine code.

During this scan compiler stores the identifiers of that application program in the symbol table. These identifiers are stored in the form of name, value address, type.

Here the name represents the name of identifier, value represents the value stored in an identifier, the address represents memory location of that identifier and type represents the data type of identifier.
Thus compiler can keep track of all the identifiers with all the necessary information.

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 the compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, a pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Operations on Symbol Table :

Following operations can be performed on symbol table-

1. Insertion of an item in the symbol table.
2. Deletion of any item from the symbol table.
3. Searching of desired item from symbol table.

Implementation of Symbol table –

Following are commonly used data structures for implementing symbol table:-

1. **List –**

   we use a single array or equivalently several arrays, to store names and their associated information ,New names are added to the list in the order in which they are encountered . The position of the end of the array is marked by the pointer available, pointing to where the next symbol-table entry will go. The search for a name proceeds backwards from the end of the array to the beginning. when the name is located the associated information can be found in the words following next.

id1	info1	id2	info2	…….	id_n	info_n

- In this method, an array is used to store names and associated information.
- A pointer **“available”** is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from the beginning of the list till available pointer and if not found we get an error **“use of the undeclared name”**
- While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e. **“Multiple defined names”**
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- The advantage is that it takes a minimum amount of space.

I. **<u>Linked List</u> –**

- This implementation is using a linked list. A link field is added to each record.
- Searching of names is done in order pointed by the link of the link field.
- A pointer **“First”** is maintained to point to the first record of the symbol table.
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

II. **<u>Hash Table</u> –**

- In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
- A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
- To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
- Insertion and lookup can be made very fast – O(1).
- The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

III. **<u>Binary Search Tree</u> –**

- Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of the root node that always follows the property of the binary search tree.
- Insertion and lookup are O($\log_2$ n) on average.