MODULE - 3

Prof. Deepali Raikar

7

Run Time Storage Organization

During execution of the input source program some data objects (those which are represented by variables) from input source text are important factors to the code generation. In the code generation these data objects are referred by their addresses. The addresses to these data objects are dependent upon the organization of memory. And the organization of memory is driven by the source language features. In this chapter we will focus on some source language issues. We will discuss the organization of memory and what are the allocation strategies that need to be adopted. Finally we will end up out discussion by symbol table organization and management.

7.1 Source Language Issues

There are various language features that affect the organization of memory. The organization of data is determined by answering the following questions. The source language issues are -

- **Does the source language allow recursion?**

  While handling the recursive calls there may be several instances of recursive procedures that are active simultaneously. Memory allocation must be needed to store each instance with its copy of local variables and parameters passed to that recursive procedure. But the number of active instances is determined by run time.

- **How the parameters are passed to the procedure?**

  There are two methods of parameter passing : call by value and call by reference. The allocation strategies for each of these methods are different.

  Some languages support passing of procedures itself as parameter or a return value of the procedure itself could be a procedure.

- **Does the procedure refer non local names? How?**

  Any procedure has a access to its local names. But a language should support the method of accessing non local names by procedures.

- **Does the language support the memory allocation and deallocation dynamically ?**

The dynamic allocation and deallocation of memory brings the effective utilization of memory.

There is a great effect of these source language issues on run time environment.

7.2 Storage Organization

7.2.1 Sub Division of Run Time Memory

- The compiler demands for a block of memory to operating system. The compiler utilizes this block of memory for running (executing) the compiled program. This block of memory is called run time storage.
- The run time storage is subdivided to hold code and data such as
  i) The generated target code
  ii) Data objects
  iii) Information which keeps track of procedure activations.
- The size of generated code is fixed. Hence the target code occupies the statically determined area of the memory. Compiler places the target code at the lower end of the memory.

The amount of memory required by the data objects is known at the compiled time and hence data objects also can be placed at the statically determined area of the memory. Compiler prefers to place the data objects in the statically determined area because these data objects then can be compiled into target code. For example, in FORTRAN all the data objects are allocated statically. Hence the static data area is on the top of code area. The subdivision of run time memory as shown by following Fig. 7.1.

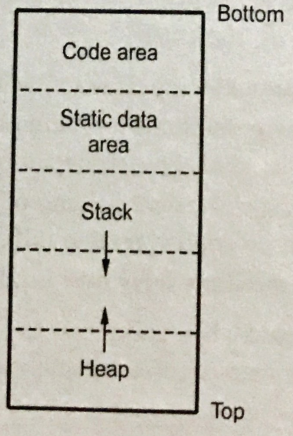


Fig. 7.1 Run time storage organization

- The counterpart of control stack is used to manage the active procedures. Managing of active procedures means that when a call occurs then execution of activation is interrupted and information about status of the stack is saved on the stack. When the control returns from the call this suspended activation is resumed after storing the values of relevant registers. Also the program counter is set to the point immediately after the call. This information is stored in the stack area of run time storage. Some data objects which are contained in this activation can be allocated on the stack along with the relevant information.
- The heap area is the area of run time storage in which the other information is stored. For example, memory for some data items is allocated under the program control. Memory required for these data items is obtained from this heap area. Memory for some activation is also allocated from heap area.
- The size of stack and heap is not fixed it may grow or shrink interchangeably during the program execution.
- Pascal and C need the run time stack.

7.3 Activation Record

- The activation record is a block of memory used for managing information needed by a single execution of a procedure.
- FORTRAN uses the static data area to store the activation record. Where as in PASCAL and C the activation record is situated in stack area. The contents of activation record are as shown in the following Fig. 7.2.

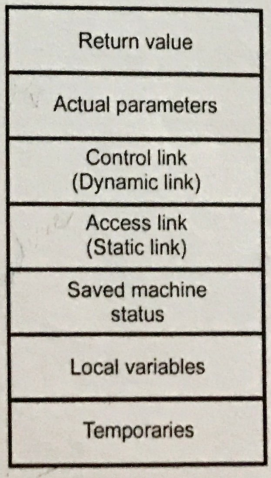


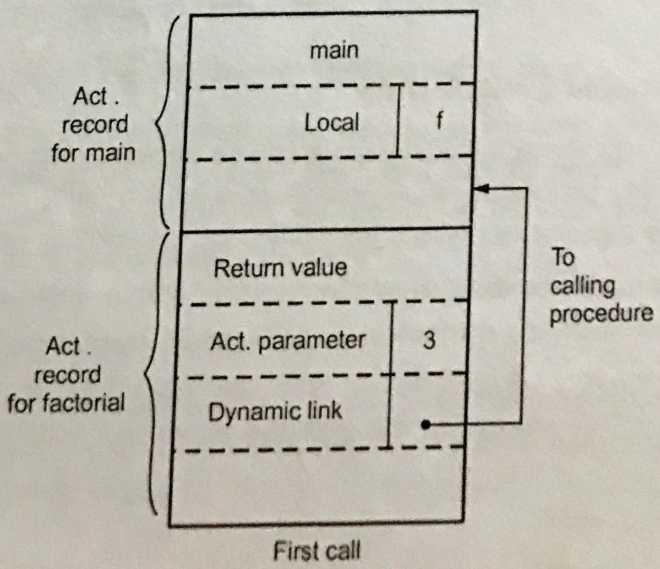
Fig. 7.2 Model of activation record

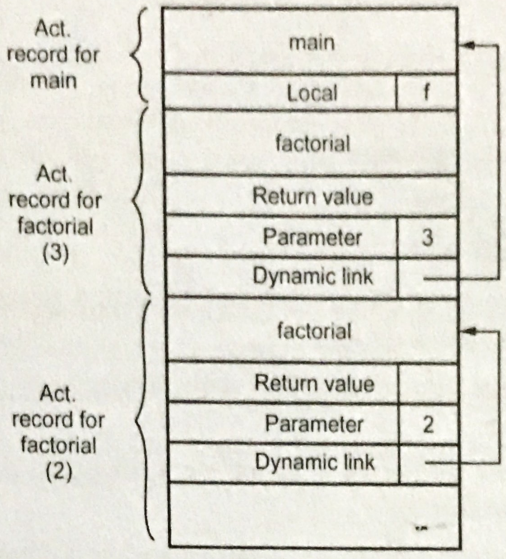
- Various fields of activation record are as follows -
  1. **Temporary values** - The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
  2. **Local variables** - The local data is a data that is local to the execution of procedure is stored in this field of activation record.
  3. **Saved machine registers** - This field holds the information regarding the status of machine just before the procedure is called. This field contains the machine registers and program counter.
  4. **Control link** - This field is optional. It points to the activation record of the calling procedure. This link is also called dynamic link.
  5. **Access link** - This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
  6. **Actual parameters** - This field holds the information about the actual parameters. These actual parameters are passed to the called procedure.
  7. **Return values** - This field is used to store the result of a function call.
- The size of each field of activation record is determined at the time when a procedure is called.

Example 7.1 : *By taking the example of factorial program explain how activation record will look like for every recursive call in case of factorial (3).*

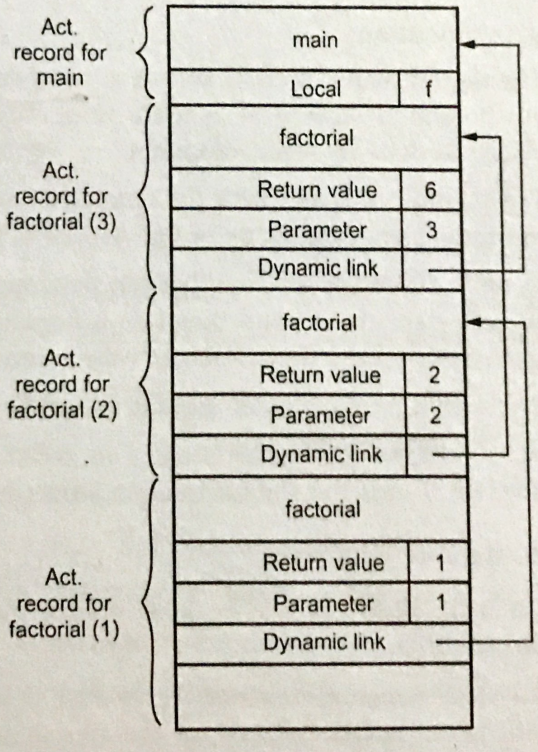
Solution :

```
main()
{
     int f;
     f = factorial (3);
}
int factorial (int n)
{
     if (n==1)
          return 1;
     use
          return (n*factorial (n-1));
}
```





Second call



Third call

7.4 Storage Allocation Strategies

As we have already discussed that the run time storage is divided into

1. Code area
2. Static data area
3. Stack area
4. Heap area

There are three different storage allocation strategies based on this division of run time storage. The strategies are -

1. **Static allocation** - The static allocation is for all the data objects at compile time.
2. **Stack allocation** - In the stack allocation a stack is used to manage the run time storage.
3. **Heap allocation** - In heap allocation the heap is used to manage the dynamic memory allocation.

Let us discuss each of these strategies in detail

7.4.1 Static Allocation

1. The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
2. The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.
3. In static allocation the compiler can determine the amount of storage required by each data object. And therefore it becomes easy for a compiler to find the addresses of these data in the activation record.
4. At compile time compiler can fill the addresses at which the target code can find the data it operates on.
5. FORTRAN uses the static allocation strategy.

Limitations of static allocation

1. The static allocation can be done only if the size of data object is known at compile time.
2. The data structures cannot be created dynamically. In the sense that, the static allocation cannot manage the allocation of memory at run time.
3. Recursive procedures are not supported by this type of allocation.

7.4.2 Stack Allocation

1. Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called **control stack**.
2. As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.
3. The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.
4. The data structures can be created dynamically for stack allocation.

Limitations of stack allocation

The memory addressing can be done using pointers and index registers. Hence this type of allocation is slower than static allocation.

7.4.3 Heap Allocation

1. If the values of non local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last In First Out nature. For retaining of such local variables heap allocation strategy is used.
2. The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object. This allocated memory can be deallocated when activation ends. This deallocated (free) space can be further reused by heap manager.
3. The efficient heap management can be done by,
   i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.
   ii) Allocate the most suitable block of memory from the linked list. i.e. use best fit technique for allocation of block.

7.5 Variable Length Data

For a procedure call sequence can be implemented by caller and callee procedures. The caller procedure is a procedure a which calls another procedure whereas the callee is a procedure which gets called by the another procedure.

For understanding, how the variable length data is handled by run time storage consider following a scenario.

Suppose a procedure A calls the procedure B. Procedure A has two arrays x and y. The storage to these arrays is not the part of activation record of A. In the

activation record of A only the pointers to the beginning of x and y are appearing. We can obtain the relative addresses of these arrays at compile time.

The activation record for the procedure B (callee procedure) begins after the arrays of A. Suppose the procedure B has variable length arrays p and q. Then after the activation record of B the arrays for procedure B can be placed.

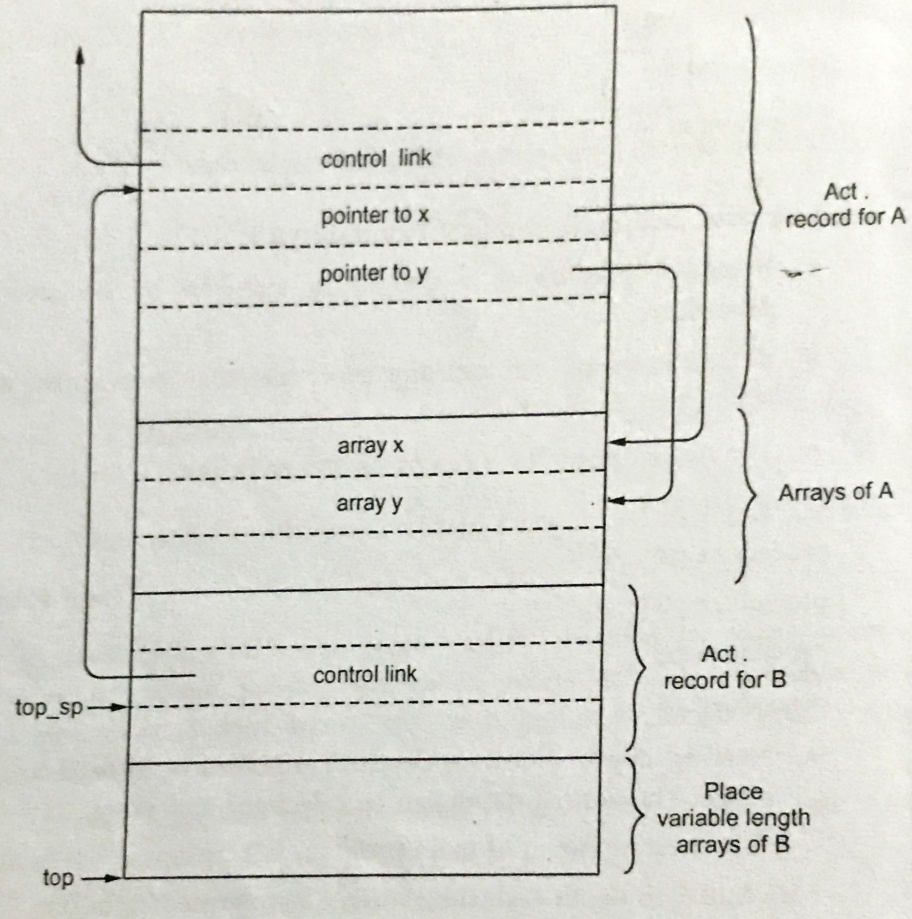


Fig. 7.3 Storing variable length data

Two pointers can be maintained top and top_sp to keep track of these records. The **top** points to the actual top of the stack and **top_sp** points to the end of some field in the activation record. By this the length of the activation record of callee procedures is known to caller at compile time.

7.6 Access to Non Local Names

A procedure may sometimes refer to variables which are not local to it. Such variables are called as non local variables. For the non local names there are two types of scope rules that can be defined.

1. Static scope rule

The static scope rule is also called as lexical scope. In this type the scope is determined by examining the program text. PASCAL, C and ADA are the languages that use the static scope rule. These languages are also called as block structured languages.

2. Dynamic scope rule

The dynamic scope rule determines the scope of declaration of the names at run time by considering the current activation.

LISP and SNOBOL are the languages which use dynamic scope rule.

In this section we will first discuss the static scope rule. First of all let us understand few terminologies.

Block

The block is a sequence of statements containing the local data declarations and enclosed within the delimiters.

For example

```
{
Declaration statements
...
}
```

The delimiters mark the beginning and end of the block. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1.

7.6.1 Static Scope or Lexical Scope

. The scope of declaration in a block structured language is given by most closely nested loop or static rule. It is as given below.

The declarations are visible at a program point are,

i) The declarations that are made locally in the procedure.

ii) The names of all enclosing procedures.

iii)The declarations of names made immediately within such procedures.

To understand the static scope consider one example.

Example 7.2 : *Obtain the static scope of the declarations made in the following piece of code.*

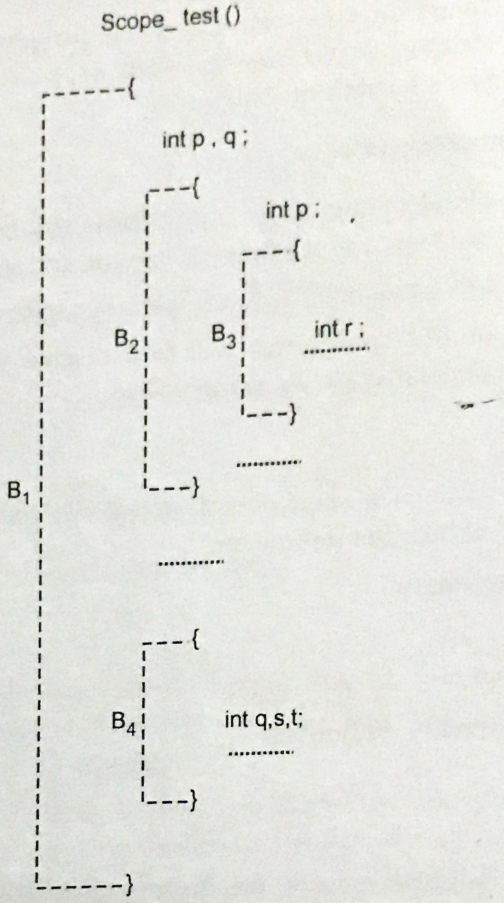
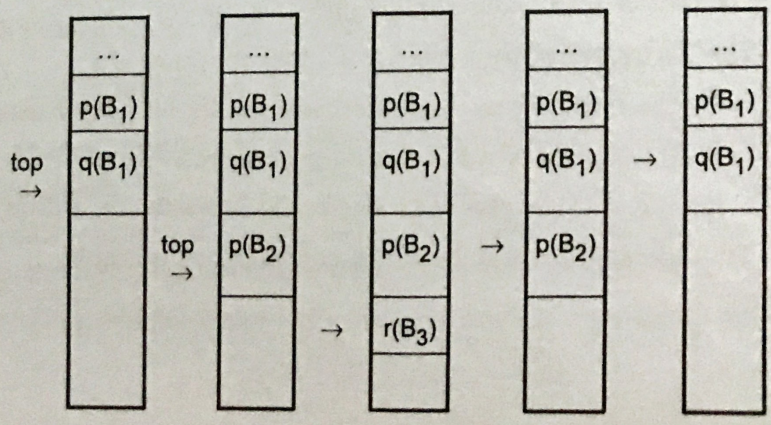
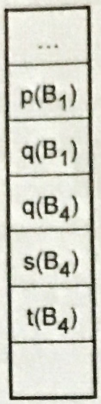


Fig. 7.4

Solution : The storage can be allocated for a complete procedure body at one time.

The storage for the names corresponding to particular block can be as shown below.





Thus block structure storage allocation can be done by stack.

7.6.2 Lexical Scope for Nested Procedures

- **Nested procedure** is a procedure that can be declared within another procedure.
- A procedure pi, can call any procedure that is its direct ancestor or older siblings of its direct ancestor.
- The nested procedures can be as shown below.

Procedure main

procedure p1

procedure p2

procedure p3

procedure p4

- **Nesting depth** - Nesting depth of a procedure is used to implement lexical scope. The nesting depth can be calculated as follows.

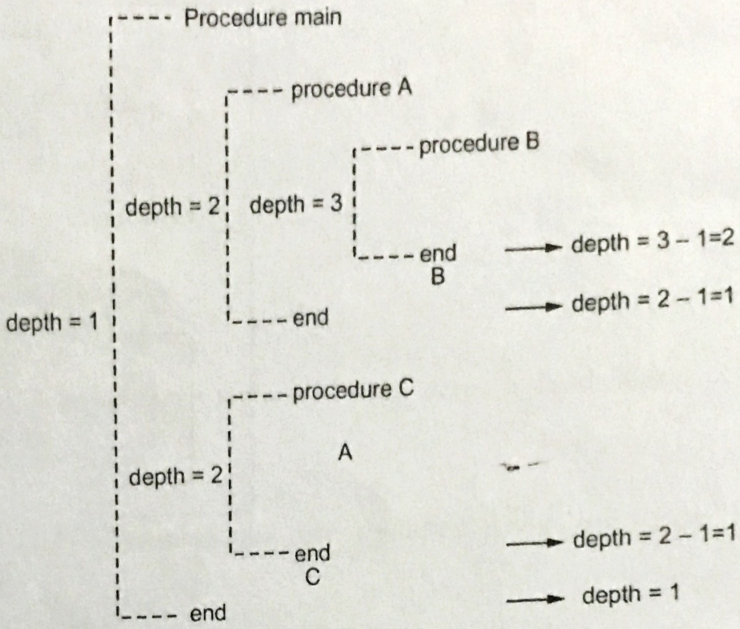
i) The nesting depth of main program is 1.

ii) Add 1 to depth each time when a new procedure begins.

iii) Subtract 1 from depth each time when you exit from a nested procedure.

iv) The variable declared in specific procedure is associated with nesting depth.

For example - Consider the following piece of code



- The lexical scope can be implemented using access link and displays.

1. Access link

The implementation of lexical scope can be obtained by using pointer to each activation record. These pointers are called access links. If procedure p is nested within a procedure q then access link of p points to access link of most recent activation record of procedure q.

For example - Consider following piece of code and the run time stack during execution of the program.

```
program  test;
    var a : int;
        procedure A;
            var d : int;
                begin a : = 1, end;
        procedure B(i : int);
            var b : int;
                procedure C;
                    var k : int ;
                    begin A; end;
                        begin
                        if (i<>0) then B(i-1)
                        else C;
                        end
                begin B(1); end;
```

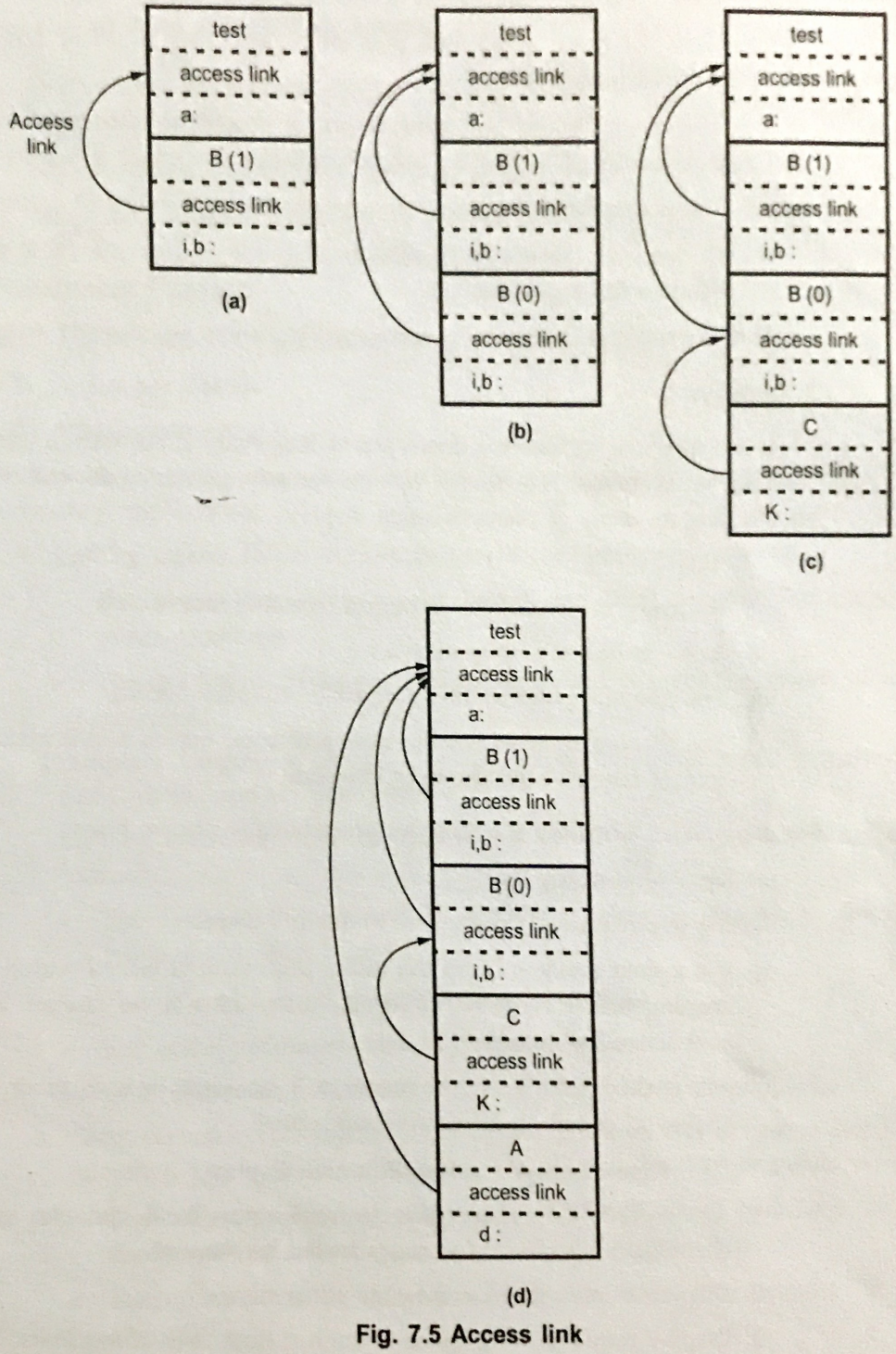


Fig. 7.5 Access link

- To set up the access links at compile time

i) If procedure A at depth n_A calls procedure B at depth n_B then

Case 1 : if $n_A < n_B$, then B is enclosed in A and $n_A = n_B - 1$.

Case 2 : if $n_A \geq n_B$, then it is either a recursive call or calling a previously declared procedure.

ii) The access link of activation record of procedure A points to activation record of procedure B where the procedure B has procedure A nested within it.

iii) The activation record for B must be active at the time of pointing.

iv) If there are several activation records for B then the most recent activation record will be pointed.

Thus by traversing the access links, non-locals can be correctly accessed.

2. Displays :

It is expensive to traverse down access link every time when a particular nonlocal variable is accessed. To speed up the access to nonlocals can be achieved by maintaining an array of pointers called display.

In display -

i) An array of pointers to activation record is maintained.

ii) Array is indexed by nesting level.

iii) The pointers point to only accessible activation record.

iv) The display changes when a new activation occurs and it must be reset when control returns from the new activation.

For example : Consider the scope of procedures as shown below -

See Fig. 7.6 on next page.

Nesting depth y calls a procedure at depth x then

i) $y < x$ then $x = y + 1$ and the called procedure is nested within caller. Then by keeping first y elements of display array as it is we can set display[x] to the new activation record.

ii) $y \geq x$ then first $x - 1$ elements will be kept as it is in display array but display[x] points to new activation record.

- **Comparison between access link and display**

1. Access link take more time to access non local variables especially when non local variables are at many nested levels away.
2. Display is used to generate code efficiently.
3. Display requires more space at the run time than access links.

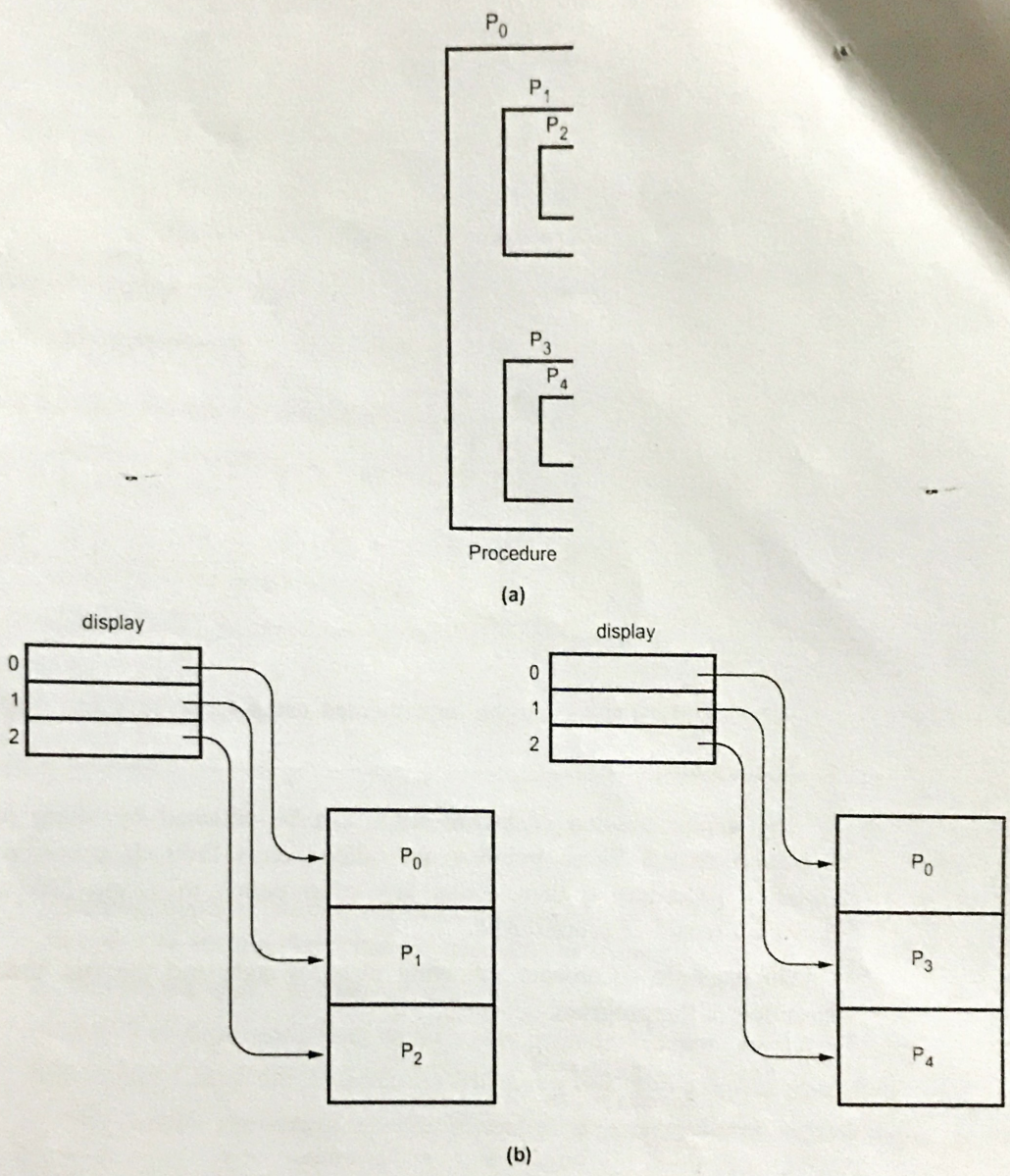


Fig. 7.6

Dynamic scoping

In dynamic scoping a use of nonlocal variable refers to the nonlocal data. Such a data is declared in most **recently called** and **still active** procedures. Therefore each time new bindings are set up for local names called procedure. In dynamic scoping symbol tables can be required at run time.

To understand the concept of dynamic scoping consider the following example.

```
procedure main()
    value : int;
procedure first ()
    begin
        value : int;
        value :=1;
        print (value);
    end
    begin
        value :=2;
        print(value);
        first();
    end;
```

In PASCAL like language static or lexical scope is used. Hence for a static scope the output will be

2 2

However, LISP like languages use dynamic scope and under dynamic scope output will be

2 1

This is because local variable 'value' is given value as 1 and global variable 'value' is given value as 2. In dynamic scoping most recent activation record is always referred. The procedure 'first' is the most recent activation record. And the variable 'value' gets value 1 over there, hence the output is 2 1.

In dynamic scoping the questions of which nonlocal variables to use is determined at run time.

There are two ways to implement nonlocal accessing under dynamic scoping.

i] Deep access

The idea is keep a stack of active variables, use control links instead of access links and when you want to find a variable then search the stack from top to bottom looking for most recent activation record that contains the space for desired variables. This method of accessing nonlocal variables is called deep access.

As search is made "deep" in the stack hence the method is called deep access.

In this method a symbol table needs to be used at run time.

ii] Shallow access

The idea is to keep a central storage with one slot for every variable name. If the names are not created at run time then that storage layout can be fixed at compile time otherwise when new activation of procedure occurs, then that procedure changes the storage entries for its locals at entry and exit (i.e. while entering in the procedure and while leaving the procedure).

Comparison of deep and shallow access

- Deep access takes longer time to access the nonlocals while shallow access allows fast access.
- Shallow access has a overhead of handling procedure entry and exit.
- Deep access needs a symbol a table at run time.

7.7 Parameter Passing

There are two types of parameters.

i) Formal parameters

ii) Actual parameters

And based on these parameters there are various parameter passing methods, the most common methods are,

1. **Call by value :** This is the simplest method of parameter passing.
   - The actual parameters are evaluated and their r-values are passed to called procedure.
   - The operations on formal parameters do not changes the values of actual parameter.

   Example : Languages like C, C++ use actual parameter passing method. In PASCAL the non-var parameters.
2. **Call by reference** : This method is also called as call by address or call by location.
   - The L-value, the address of actual parameter is passed to the called routines activation record.
   - The values of actual parameters can be changed.
   - The actual parameters should have an L-value.

   Example : Reference parameters in C++, PASCAL'S var parameters.
3. **Copy restore** : This method is a hybrid between call by value and call by reference. This method is also known as copy-in-copy-out or values result.
   - The calling procedure calculates the value of actual parameter and it is then copied to activation record for the called procedure.
   - During execution of called procedure, the actual parameters value is not affected.
   - If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

Example : In Ada this parameter passing method is used.

4. **Call by name :** This is less popular method of parameter passing.
   - Procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.
   - The actual parameters can be surrounded by parenthesis to preserve their integrity.
   - The locals names of called procedure and names of calling procedure are distinct.

Example : ALGOL uses call by name method.

For example : Consider the following piece of code along with the output.

```
procedure exchange (m,n:integer);
    var t : integer;
    begin
        t := m;
        m := n;
        n ;= t;
    end;

    ...
    i ;= 1
    a[i] := 50 { a:array [1....10] of integer}
    print (i,a[i]);
    exchange{i,a[i]);
    print (i,a[1]);
```

The output for all the above discussed method is

Call by value		Call by reference		Copy restore		Call by name	
1	50	1	50	1	50	1	50
1	50	50	1	50	1	Error	

The error occurs because

t := i ∴ t := 1

i := a[i] ∴ i := 50 as a[i] = 50

a[i] := t ∴ a[50] = t = 1 then index is out of bounds.

7.8 Symbol Tables

Definition

Symbol table is a data structure used by compiler to keep track of semantics of variable. That means symbol table.

Stores the information about scope and binding information about names.

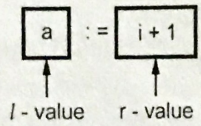
- Symbol table is buit in lexical and syntax analysis phases.

  The symbol table is used by various phases as follows - Semantic analysis phase refers symbol table for type conflict issue. Code generation refers symbol table knowing how much run-time space is allocated? What type of run-time space is allocated?

l-value and r-value

The *l* and r prefixes come from left and right side assignment.

For example :



7.8.1 Symbol - Table Entries

- To achieve compile time efficiency compiler makes use of symbol table.
- It associates lexical names with their attributes.
- The items to be stored in symbol table are,

i) Variable names
ii) Constants
iii) Procedure names
iv) Function names
v) Literal constants and strings
vi) Compiler generated temporaries
vii) Labels in source languages

- Compiler uses following types of information from symbol table.

i) Data type
ii) Name
iii) Declaring procedures
iv) Offset in storage

v) If structure or record then pointer to structure table

vi) For parameters, whether parameter passing is by value or reference?

vii) Number and type of arguments passed to the function

viii) Base address

7.8.2 How to Store Names in Symbol Table?

There are two types of name representation.

1. Fixed length name

A fixed space for each name is allocated in symtab. In this type of storage if name is too small then there is wastage of space.

For example :

Name										Attribute
c	a	l	c	u	l	a	t	e		
s	u	m								
a										
b										

The name can be referred by pointer to symbol table entry.

2. Variable length name

The amount of space required by string is used to store the names.

The name can be stored with the help of starting index and length of each name.

For example :

Name		Attribute
Starting index	Length	
0	10	
10	4	
14	2	
16	2	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$

7.8.3 Symbol Table Management

Requirements for symbol table management.

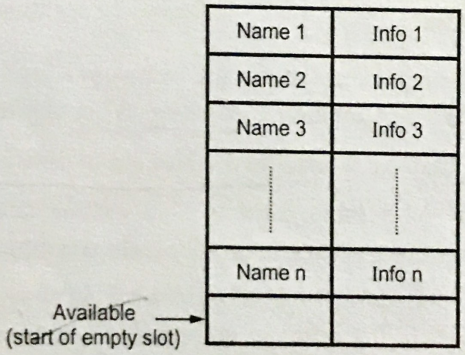
i) For quick insertion of identifier and related information

ii) For quick searching of identifier

Following are commonly used data structures for symbol table construction.

1. List data structure for symbol table

- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive.
- The pointer 'available' is maintained at the end of all stored records. The Figure for list data structure using arrays is as given below.



- 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 should not be already there. If it is there another error occurs i.e. "Multiple defined Name".
- The advantages of list organization is that it takes minimum amount of space.

2. Self organizing list - using linked list

- This symbol table implementation is using linked list. A link field is added to each record.

- We search the records in the order pointed by the link of link field.
- A pointer "First" is maintained to point to first record of the symbol table.

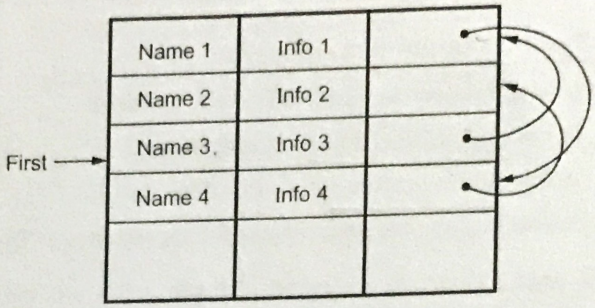


Fig. 7.7 Symbol table

The reference to these names can be Name 3, Name 1, Name 4, Name 2.

- When the name is referenced or created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

3. Hash tables

- Hashing is an important techniques used to search the records of symbol table. This method is superior to list organization.
- In hashing scheme two tables are maintained a hash table and symbol table.
- The hash table consists of k entries from 0, 1 to k–1. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that h(name) will result any integer between 0 to k–1. We can search any name by

  position = h(name)

- Using this position we can obtian the exact locations of name in symbol table.
- The hash table and symbol table can be as shown below.

See Fig. 7.8 on next page

- The hash function should result in uniform distribution of names in symbol table.
- The hash function should be such that there will be minimum number of collision. (Collision is such a situation where hash function results in same location for storing the names.)

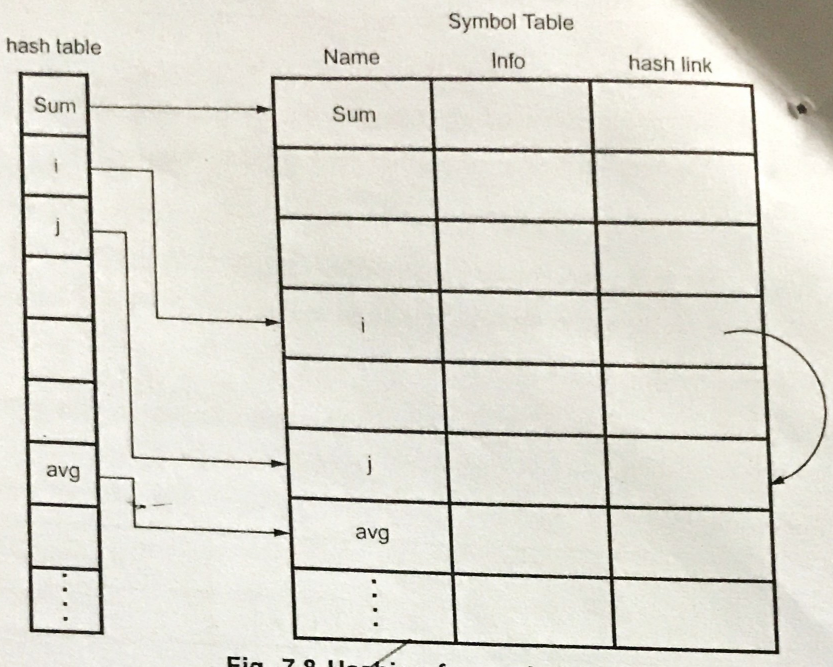


Fig. 7.8 Hashing for symbol table

- Various collision resolution techniques are open addressing, chaining, rehashing.
- The advantage of hashing is quick search is possible and the disadvantage is that hashing is complicated to implement. Some extra space is required. Obtaining scope of variables is very difficult.

7.9 Error Detection and Recovery

In this phase of compilation all possible errors made by the programmer are detected and they are reported to the user in the form of messages. This process of locating errors and reporting to user is called **error handling process.**

Functions of error handler

1. The error handler should identify all possible errors from the source code.
2. It should report these errors with appropriate messages to the user / programmer. The error messages should be informative so that the programmer can correct those.
3. The error handler should repair the errors at that instance in order to continue processing of the program.
4. The error handler should recover from errors in order to detect as many errors as possible.

7.10 Classification of Errors

The errors can be classified as

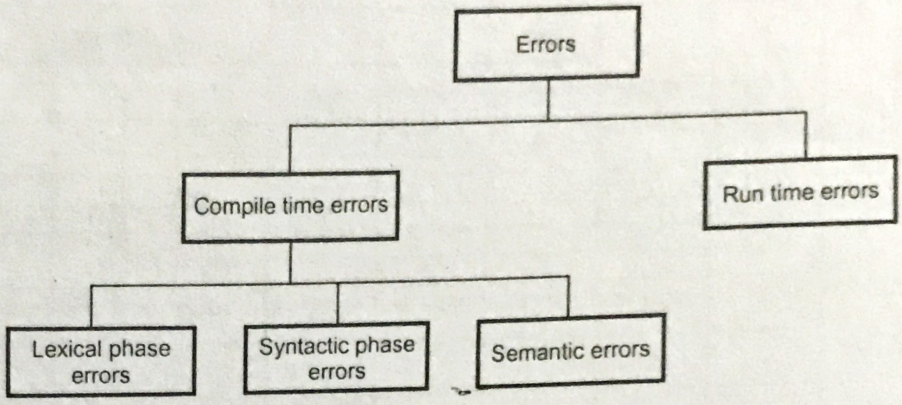


Fig. 7.9 Classification of errors

Globally there are two types of errors : Compile time and run time errors.

During the phases of compiler these errors can be potentially detected hence they are classified by names of compilation phases.

7.10.1 Lexical Phase Errors

These type of errors can be detected during lexical analysis phase. Typical lexical phase errors are

- Spelling errors. Hence get incorrect tokens.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.

Normally due to typing mistakes the wrong spellings may appear in the program. This causes a lexical error.

For example

```
swtich (choice)
{
.....................
.....................
}
```

In above code 'switch' cannot be recognized as a misspelling of keyword switch. Rather lexical analyzer will understand that it is an identifier and will return it as valid identifier. Thus misspelling causes errors in token formation. This is can be a possibility of an error which occurs due to cascading of characters.

For example

```
switch (choice)
{
case1:   get_data();
         break;
case2:   display();
         break;
}
```

In above given 'C' code case 1 will not be identified because there is no space between 'case' and '1'. But the compiler does not report any error message as it identifies 'case1' as valid identifier. This type of error is lexical phase error.

Consider

```
printf("\n Hello India");$
```

This is also a lexical error as an illegal character '$' appears at the end of the statement.

If length of identifiers get exceeded the error occurs. For example, if in FORTRAN there is an identifier whose length is of 10 characters then lexical error occurrs.

7.10.2 Syntactic Phase Errors

These are the errors which are popularly known as syntax errors. They appear during syntax analysis phase of compiler.

Typical phase errors are :

- Errors in structure
- Missing operators
- Unbalanced parenthesis.

The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.

If the error situations are explicitly known, reporting of errors is not difficult. The error reporting is usually done with appropriate error messages. In table driven parsers like LL(1), SLR, canonical LR and LALR parsers explicit entries for error situations are given. The error detection and reporting in a table driven parser can be done as follows.

Consider the grammar -

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

The SLR parsing table for this grammar is

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

First approach

The error entries for each state are denoted by e_i.

1. Create an error entry for state i denoted by e_i. This entry will be invoked when we receive the corresponding entry in state i.
2. Let T be the set of valid tokens. Valid token means Action [i, t] ≠ error where $t \in T$.
3. The routine e_i issues an error message as "expected symbols are T".

For example - Consider state 2 of above given SLR parsing table. We can create an entry for error routine e2 in state 2 in blank slots.

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2	e2	r2	s7	e2	r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The routine e2 will generate an error message as :

Error at line ____ : expected symbols are +, *,) and end of input.

The next_token will give the line number.

Second approach

As e2 indicates that there are two erroneous tokens 'id' and '('.

We will find 'less likely candidates' and 'more likely candidates' from set of tokens T.

If id comes after + or * then it becomes ' ... + id' or ' ... * id'. We will consider the scenario as :

Stack	Input	Moves
0T2	+ id	Reduce E → T
0E1	+ id	Shift
0E1 + 6	id	Shift
0E1 + 6id 5	...	...

The tokens get processed without error. Similarly it can be processed for * id as well. This shows that + and * are $ more likely candidates as 'tid' gets parsed without error.

Now consider) id

Stack	Input	Moves
0T2	) id	Reduce E → T
0E1	) id	Error

Thus tid with t =) does not parse the input. Hence) is called less likely candidate. Hence now we can change the error message given by e2 to.

"Error at line _____ : Missing operator".

Third approach

In this approach of error handling we try to reduce the error entries from parsing table. Following are the rules which are used to reduce error entries.

1. If the state has one reduce entry r_j then all error entries of state are replaced by r_j.
2. If there are more than one frequent reduce entries then the most frequently used reduce entry is chosen for replacement.
3. The state having shift entries only will have their error entries as it is.

For example - Consider parsing table with reduced error entries.

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2	r2	r2	s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5			s4			8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9	r1	r1	s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

The above table has now reduced error entries. Each blank slot in above parsing table can be filled up with appropriate error routine. It can be as shown -

State	id	+	*	(	)	$	E	T	F
0	s5	e1	e1	s4	e2	e1	1	2	3
1	e3	s6	e3	e3	e2	Accept			
2	r2	r2	s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5	e1	e1	s4	e2	e1	8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5	e1	e1	s4	e2	e1		9	3
7	s5	e1	e1	s4	e2	e1			10
8	e3	s6	e3	e3	s11	e1			
9	r1	r1	s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

The error routines will generate following error messages -

Error routine	Messages
e1	Missing operand
e2	Right parenthesis not matching
e3	Missing operator
e4	Missing right parenthesis

7.10.3 Semantic Errors

Semantic errors are those errors which get detected during semantic analysis phase.

Typical errors in this phase are :

- Incompatible types of operands.
- Undeclared variables.
- Not matching of actual arguments with formal arguments.