24th April '17
Lecture-55

Unit-4

Symbol Tables

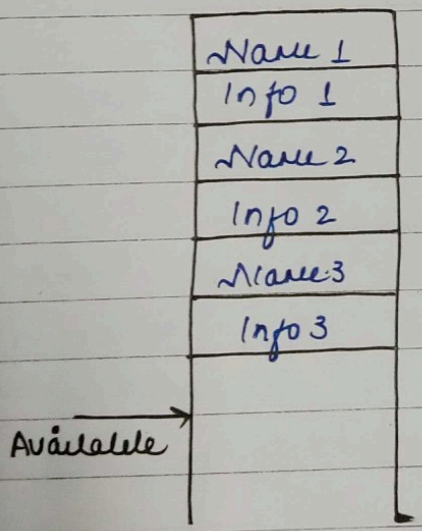
A compiler needs to collect & use information about the names appearing in the source program. This information is entered into a datastructure called Symbol Table.

* Datastructures for Symbol Tables:

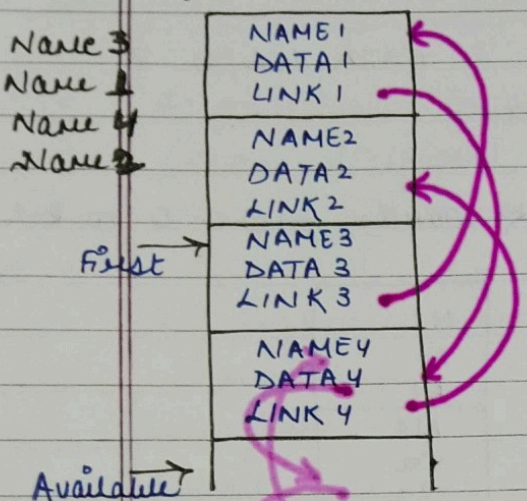
I Lists:

We use a single or equivalently several arrays to store names & their associated information.

Examples



2 Self Organizing List :



- In this more space is used.
- It is more efficient than List.
- It consist of 3 fields :
  1.) Name - : consists of name
  2.) Data : consists of data
  3.) Link : consists the name of the next name to be called

Ex : First, NAME 3 called & in LINK 3 add of 1st block is stored so NAME 1 is called then NAME 4 & lastly NAME 2.

3. Search Tree :

- A more efficient approach to symbol A table organization is to add two linked fields : LEFT & RIGHT to each record.

- We use these fields to link the records in a BST.

- If we are searching for NAME & have found the record for NAME$_i$ we need only follow LEFT$_i$ if NAME < NAME$_i$ and need only follow RIGHT$_i$ if NAME > NAME$_i$.

- Algorithm :

1.) While P ≠ NULL do
2.) if NAME = NAME (P) then NAME found
3) else if NAME < NAME (P) then P := LEFT (P)

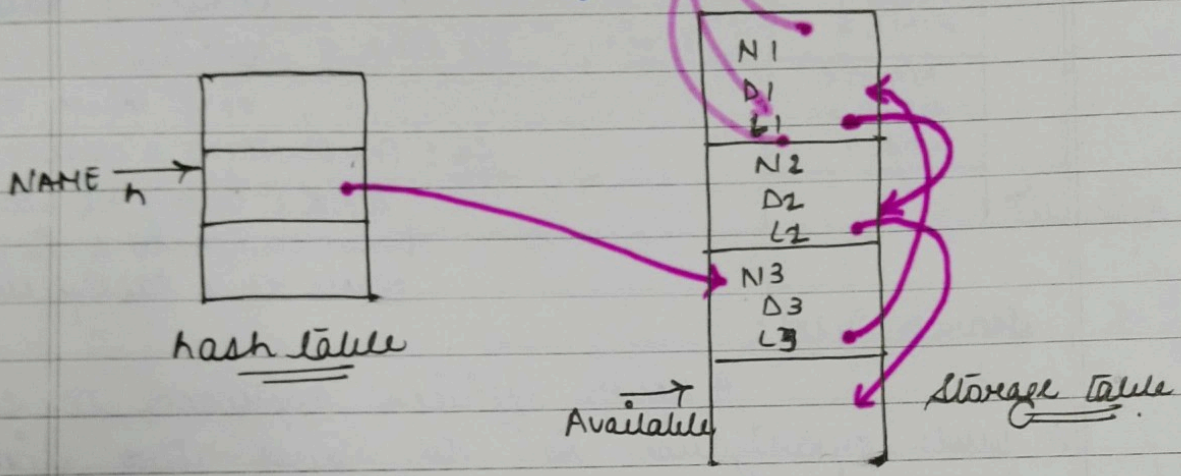
4) else NAME > NAME(P) then P := RIGHT(P).

25th April '17
Lecture 36

4. Hash table :

2 tables : Hash tables & storage tables are used.

Hash table consists of k- words. from 0 to k-1.



These words are pointers into the storage table to the heads h(name). of seperate linked list.

To determine whether NAME is in Symbol table we apply to NAME a hash function "h", such that h(name) is an integer b/w 0 to k-1.

To enter NAME into symbol table we create a record for it at the 1st available place in the storage table and link that record to the beginning of the h(NAME*) list.

25th April '17
Lecture 36

Run-time environment / administration :

A program as a source code is collection of text (code, statement) and to make it alive it requires actions to be performed on target machine.

A program needs memory resources to execute instructions. A program contains name 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 the state of the target machine which may include s/w libraries, environment variable etc. to provide services to the processes running in the system.

Activation record :

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially.

The execution of a procedure is called is activation. An activation record contains all the necessary information required to call a procedure.

An activation record may contain following units / field :

1.) Temporaries : Stores temporary & intermediate value of an expression.

Ex : $x = y + z$

$t_1 = y + z$

$x = t_2$

temp.

2.) Local data : Store local data of the called procedure.

3) Machine Status : Stores M/C status such as registers, PC (program counter) etc before procedure is called.

4.) Control Link : Stores the address of the activation record of the caller procedure.

5) Access Link : Stores the information of data which is outside the local scope

6.) Actual Parameter : Stores actual parameter i.e; procedure which are used to send i/p to the called procedure.

7.) Return value: Stores return value.

Whenever a procedure is executed its activation record is stored on the stack also known as control stack.

When a procedure call another procedure the execution of the caller is suspended until the called procedure finish execution.

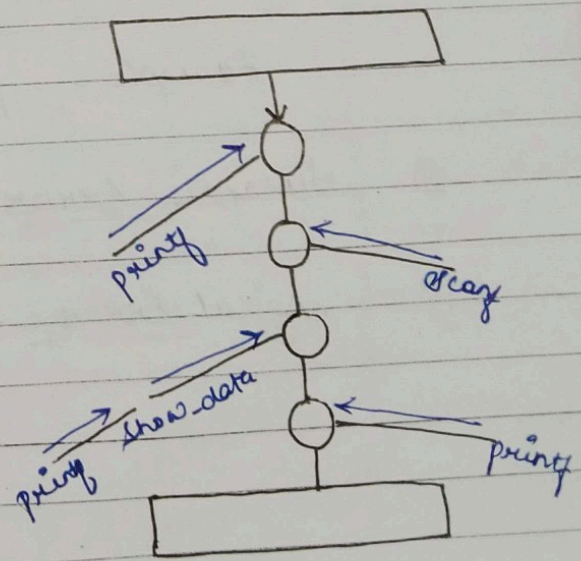
At the time the activation record of the called procedure is stored on the stack.

We assume that the program control flow in a sequential manner & when a procedure is called its control transfer to called procedure. This type of control flow makes it easier to represent a series of activation in the form of tree known as Activation Tree.

Example:

```
printf("—")
scanf("—")
show_data(—)
printf("—")

show_data()
{
    printf
}
```



26th April '17
Lecture-57

Error detection & Recovery :

Parser should be able to detect & report any error in the program. It is expected that when an error is encountered the parser should be able to handle it & carry out parsing the rest of the input.

A program may have the following kinds of errors at various stages / phases :

I. Lexical Error : Name of some identifiers typed incorrectly.

Exs

```
int (x);                         prinf( " -- " );
{                                 must be printf
   =
   printf((x1)); ← must be x
}
```

II. Syntactical Errors : Missing semicolon or unbalance paranthesis

Example : printf("Hello") ";" missing

```
int x;
}
  --
missing { }
```

III Semantic Error : Incompatible value assignment

IV Logical Error : Code not reachable , infinite loops

There are 4 common error recovery strategies that can be implemented:

1. Panic mode: When a parser encounters an error anywhere in the statement it ignores the rest of the statement by not processing input from erroneous input to delimiter such as ";".

This is the easiest way of error recovery & also it prevents the parser from developing ∞ loops.

2. Statement mode: When a parser encounters an error it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.

Ex: Inserting a missing ";", replacing ":" with ";" etc.

Parser designers have to be careful because one wrong correction may lead to an ∞ loop.

3. Error productions: Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used as productions that generate erroneous constructs when these errors are encountered.

4. Global correction: The parser considers the program as a whole & tries to figure out what the program is intended to do and tries to find out a closest match for it which is error-free.

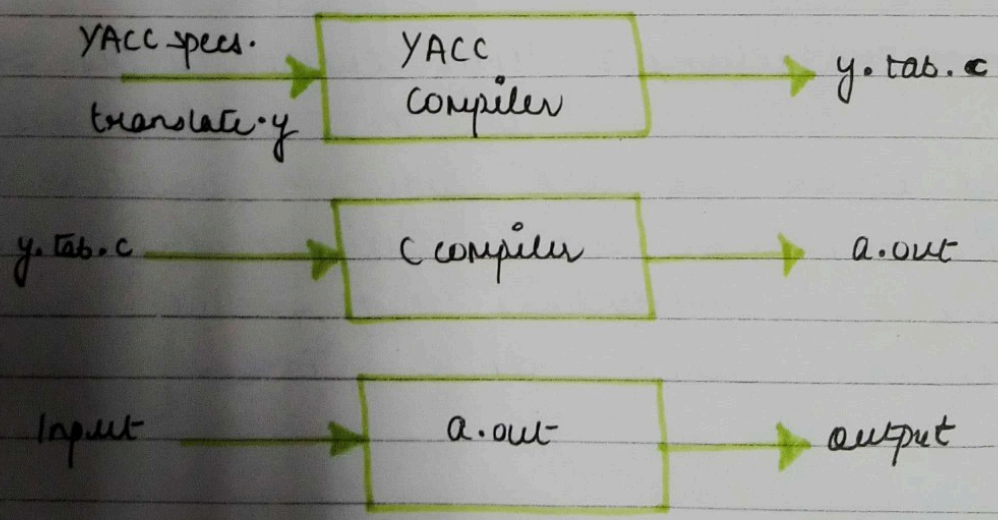
When an error is input, X is fed. It creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code.

Parser Generators:

YACC (Yet Another compiler - compiler):

→ YACC is a LALR parser generator which is used to generate a parser program.

→ YACC is available as a command on the UNIX system & has been used to help implement many production compilers.



Date: —/—/—
Page:

→ First a file **translate.y** containing a YACC specs. of the translation is prepared. The UNIX system command YACC translate.y translates the file translate.y into a C program called y.tab.c using LALR method.

→ The program y.tab.c is a representation of a LALR parser written in C along with other C routines that the user may have prepared.

→ By compiling y.tab.c along with the ly library that contains the LR parsing program using the command, "cc y.tab.c -ly" we obtain the desired object program a.out that performs the translation specified by the original YACC program.

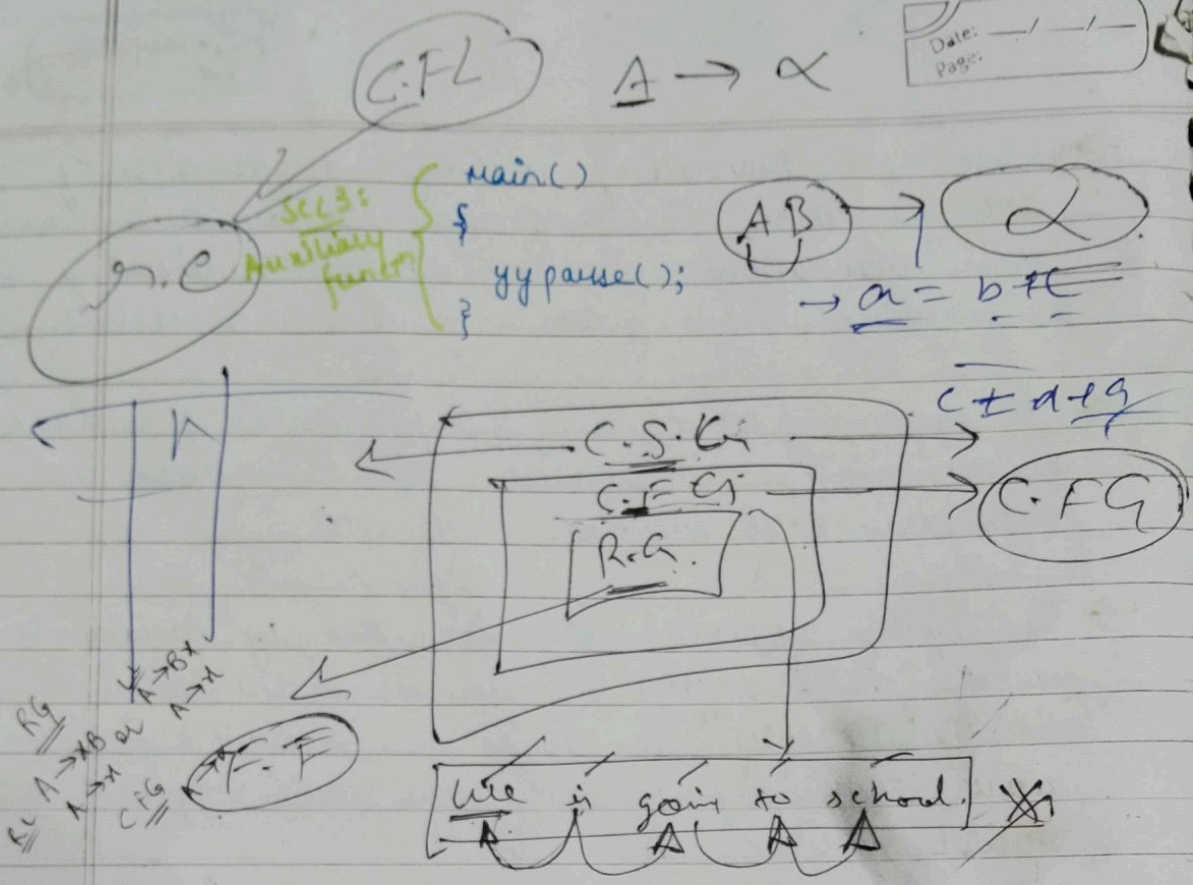
→ A YACC source program has 3 parts (sections):

* Section 1: declarations
  %%
* Section 2: translation rules
  %%
* Section 3: Auxiliary funct^n / Supporting C routines

Example: S → A|B

Sec-1 Declaration:
```
%{
  #include<stdio.h>
%}
% Start S
% token A,B
%%
```

Sec-2 translation rule:
```
S: A {Semantic action}
 | B {Semantic action}
 ;
%%
```



a + b

1, 2, 3, 4

1.) Describe symbol table & its entries. Also discuss various data structures used for symbol table

2.) What are lexical phase errors, syntactic phase errors & semantic phase errors. with suitable examples.

3.) Why run time storage management is required. How simple stack implementation is implemented.