Yacc Programming

By Dr. M M Math, Professor , Department of CSE, GIT, Belagavi

A YACC Tool

I. Introduction.
II. Structure of YACC program Specification
III. Prerequisite for Writing YACC program
IV. Running YACC program.

I. Introduction

- **What is YACC ?**
  - YACC is a programming tool on UNIX platform that takes YACC input file specification (which are set of descriptions of **syntactic constructs** in the form **of Grammar**.) and generates a **C-routines (y.tab.c)** which is termed as ***syntax analyzer or parser***. It also generates a token definitions in a separate file called **as y.tab.h**
- **Syntactic Construct :** Are nothing but the programming statement of any programming language that includes arithmetic statement, declaration statement, branching statements , Looping statements etc….
- **GRAMMER :** It is a powerful mechanism to describe the structure of statement/sentence
- **Syntax analyzer :** **I**s a program for a typical compiler that is responsible for grouping the tokens into a valid constructs, according to the specifications of grammar..
- **LEX and YACC Tools are** Written and developed by ***Eric Schmidt*** and ***Mike Lesk.***

- **Applications of YACC :**

  1. Mainly used by compiler writers to write compiler and interpreter.

II. Structure of YACC program Specification

- The Structure of YACC program specification consists of three sections, separated by a line with just %% in it:

```
%{
      SECTION -I      { Definition section }
%}
%token  { list of tokens used by YACC }
%%
      SCETION –II     { RULES Sections  }
%%
SECTION – III  { User subroutine section }
```

Definition section :

- It introduces any initial C program code like header files, declaration and initialization of variables used in the program. This is simply copied to generated C-code.
- The C program code is surrounded with two special delimiters %{ and %}
- It also contains declarations of token definitions used by the parser which is of the following form

```
%token <list of token name separated by single blank space >
```

Example:

```
%token DIGIT ID
```

NOTE : Tokens with sigle character can used directly without declarations. However these can be enclosed by single quotes

Rules section

- The rules section of the YACC program specifications contains a series of rules(productions rules) where each rule is made up single name on the left side of the ":" operator, a list symbols and action routine on the right side of the ':' operator. Each rule must be terminated with ';'

```
<Single_name>: <list of symbols>  <action routines>
      ;
```

Example:

```
stamt: expression
    ;
expression: expression '+' expression
     |      expression '-' expression
     |      DIGIT
     ;
```

- ***<action>*** is made of action routines written using C- language and are enclosed between curly braces { and }.
- It also consists of call statements to c routines written in user subroutine section of LEX specification in section III.

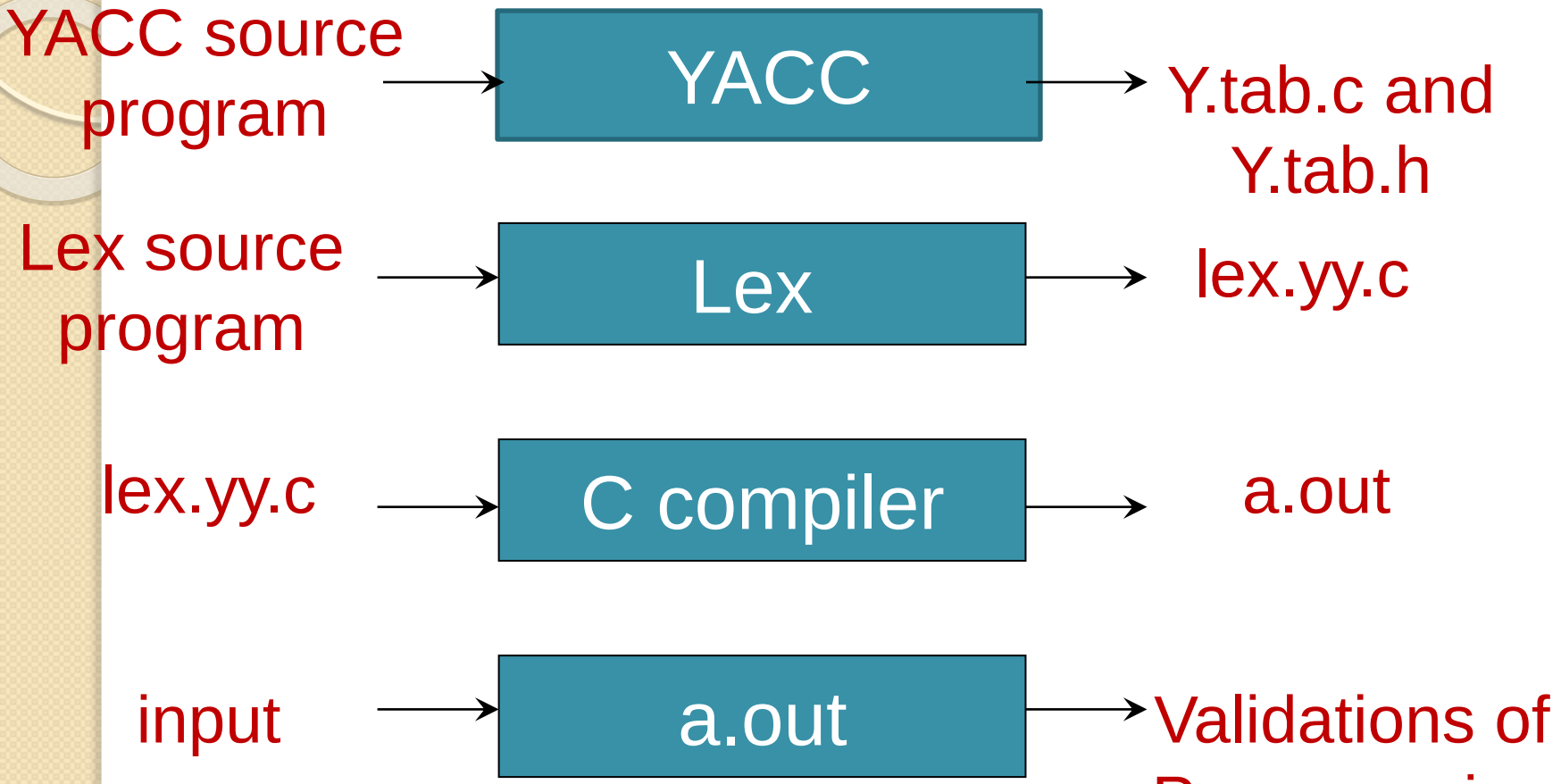
Example :

```
Statement: expression { printf(" IT is
        ;                    expression\n"); }
```

III. User Subroutine Section

- This is a final section which consists of any legal C-code that is copied to the end of the Code generated.
- It also consists of user defined c-functions which may called in the action part of the rule section.
- Finally it includes the main() function consisting of a call statement to yyparse(), where yyparse() is c-routine generated by PARESE.

Running a YACC program

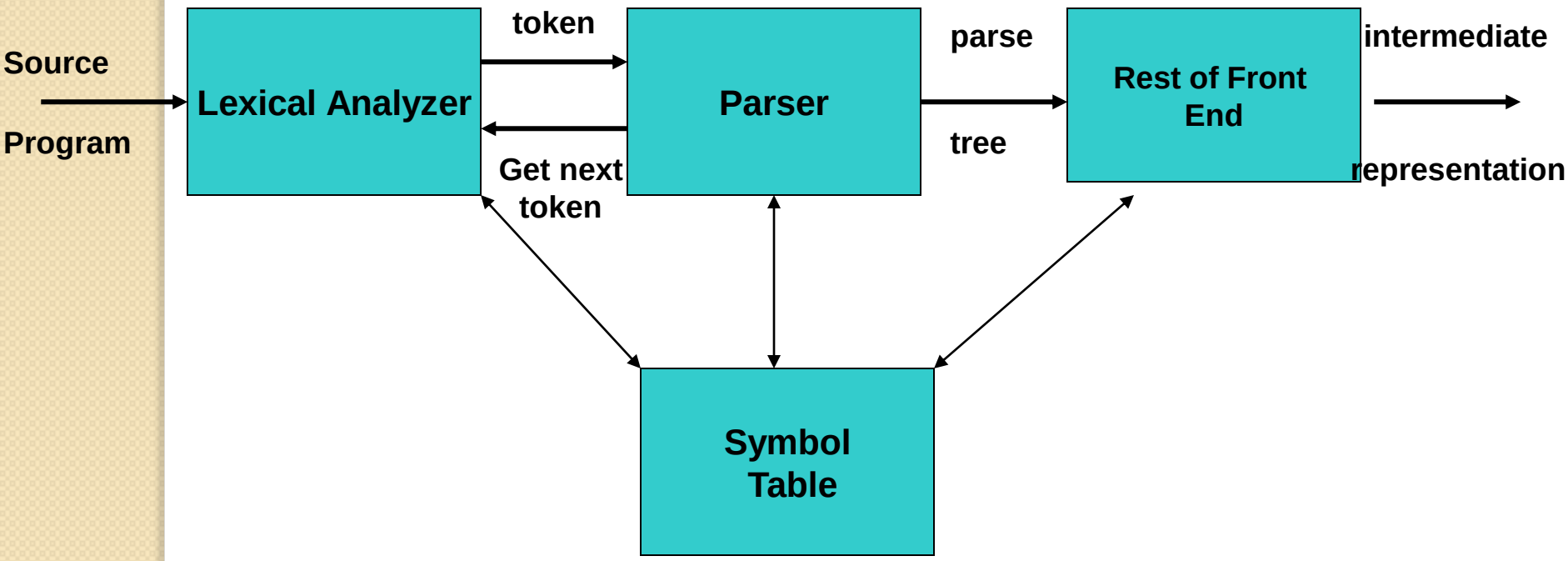



```
$yacc –d  xyz.y
$lex xyz.l
$ cc y.tab.c lex.yy.c -ly –ll
$\.a.out
```

Parser or Syntax Analyser

- Introduction and Role of the parser:
  - Syntax Analyser determines the structure of the program.
  - The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
  - Syntax Analyser uses context free grammar to define and validate rules for language construct.
  - Output of Syntax Analyser is parse tree or syntax tree which is hierarchical / tree structure of the input.
- Issues in the design
  - There is a need of mechanism to describe the structure of syntactic units or syntactic constructs of programming language. Context free grammar
  - There is a need of mechanism to recognize the structure of syntactic units or syntactic constructs of programming language. Automata.

Interaction Between Parser and Lexical Analyser.



Interaction between parser and Lexical Analyzer.

Working :

In the compilation process syntax analyser/parser is a high level routine and its responsibility is to obtain the stream of tokens from the lexical analyser and groups them into a grammatical structured as specified by the Grammar. In doing this it needs a token hence it executes a command called as getnextToken() to activate the lexical analyser. There is proper coordination between parser and the lexical analyser.

The lexical analyser then reads the source program line by line and character by character and groups the character into a meaning sequence called as tokens. These identified stream of token are returned to parser for further grouping. Any deviation in this process the error would be reported by either parser or lexical analyser. Otherwise the parser takes all the received tokens and groups them into a valid construct and reports that the construct/ source program is well formed according to the language specification.

Role of a parser.

- The stream of tokens is input to the syntax analyzer.

  The job of the parser is:

  To identify the valid statement represented by the stream of tokens as per the syntax of the language. If it is a valid statement, it will be represented by a parse tree. If it is not a valid statement, then a suitable error message is displayed, so that the programmer is able to correct the syntax error.

Shift reduce parser

- Working principle

  Shift reduce parsing attempt to construct parse tree for an input string beginning at the leaves and working up towards the root. i.e it is the process of reducing a string ' w' to the start symbol of the grammar. At each reduction step a particular substring matching with the right hand side of the production is obtained and is replaced by the symbol on the left hand side of the production. This process is continued until a string 'w' is reduced to start symbol

Note : At each step, if a correct substring is chosen then it is an exact trace of right most derivation in reverse

Algorithm of Shift Reduce parser

1. Initialize the stack to empty and input buffer holds the string 'w' to be parsed. Also input pointer is pointing to the first character of W.

   i. e. Stack Input

   $ W$

2. During the left to right scan of the input, parser shifts zero or more input symbols onto the stack until ***Handle* β is** onto the Stack.

3. Perform reduction action using the production A->β. i.e β is popped from the stack and A is pushed.

4. Repeat the steps 2 and 3 until error is detected or until the stack contains the start symbol and the input is empty.

   i. e. Stack Input

   $S $

5. Upon entering the above configuration Parser halts and announces successful completion of parsing.

Working of Shift Reduce Parser

Grammar :

S→ E

E→ E + E | E – E | E * E | E / E | DIGIT

Stack	Input	Action
$	2+5*4 $	Shift 2
$2	+5*4 $	Reduce by E→DIGIT
$E	+5*4 $	Shift +
$E+	5*4 $	Shift 5
$E + 5	*4 $	Reduce by E→DIGIT
$E + E	*4 $	Shift *
$E + E *	4$	Shift 4
$E + E * 4	$	Reduce by E→DIGIT
$E + E * E	$	Reduce by E→E * E
$E + E	$	Reduce by E→E * E
$E	$	Accept