COMPUTER ENGINEERING DEPARTMENT

ASSIGNMENT NO-03

SUB: SPCC

COURSE: T.E. Year: 2020-2021 Semester: VI
DEPT: Computer Engineering
SUBJECT CODE: CSC602 SUBMISSION DATE: 30/04/2021

==========

Name: Amey Thakur Roll No.: 50

Class: TE Comps B-50 ID: TU3F1819127

Assignment No 3

Sr. No.	Question
1	Case Study on various compiler construction tools. e.g. Lex and YACC

Amey

AMEY B 50 Amey

Page No.:
Date:

Q1 Case Study on various compiler construction tools. eg. Lex and YACC

Ans:

Compiler Construction Case Study
(Lex and YACC)

The purpose of this case study is to give an example of a compiler/interpreter front-end written in C using Lex and YACC. An interpreter is used since it allows a working program to be created with minimal extra effor (after the construction of the front end). This code could be developed into a compiler by replacing the last phase with a compiler back-end.

The code is shown in an order which underlines the processes of creating a compiler, so the same file will be shown multiple times as it developed.

The case study develops an interpreter, the following steps shall be taken to complete the interpreter.

- Lexical Analysis
- Syntax Analysis
- Lexical Analysis with Semantics
- Abstract Syntax Trees
- Generating Abstract Syntax Trees
- Interpreting.

Many important features of a useful compiler/interpreter have been left out, for brevity and simplicity, including:

- Dealing with errors
- Optimization.

Also some processes do not apply to such a simple language as very Tiny Basic. For example, Name Tables do not apply since only single character are used to name variables.

AMEY B 50 Amey

Page No.:
Date:

Requirements
- A C compiler
- Lex or equivalent (flex).
- YACC or equivalent (bison).
- (Preferred) The Make utility.

Lexical Analysis
- The first draft of the lex file identifies different tokens by returning a certain value in the associated C action.
- For keywords and operators this is simple, however identifiers, values and comments are trickier.

VTB.1 - Version 1

```
%{

%}

DIGIT [0-9]
LETTER [A-Z]

%%
"LET"          { return LET; }
"IF"           { return IF; }
"THEN"         { return THEN; }
"PRINT"        { return PRINT; }
"REM"[^\n]*    { return REM; }
"GOTO"         { return GOTO; }
"GOSUB"        { return GOSUB; }
"RETURN"       { return RETURN; }
"STOP"         { return STOP; }
"FOR"          { return FOR; }
```

AMEY B 50 Amey

Page No.:
Date:

```
"TO"          { return TO; }
"STEP"        { return STEP; }
"NEXT"        { return NEXT; }
\"[^\"]\"     { return STRING; }
"="           { return EQUAL; }
"<>"          { return NEQUAL; }
%%
```

This file can be processed using one of the following

lex VTB.l
flex VTB.l

You may wonder where all those values we returned are coming from. They will be created by YACC grammar file when it is processed.

There are some differences from the very Tiny Basic, for instance MULT and DIV in place of MULDIV. This is because we need to know the difference between the two. LineNumber and WholeNumber are lexically identical, and so cannot be separated at this time. Defining the use and category of tokens is left until the next stage.

AMEY B 50 Amey

Page No.:
Date:

Syntax Analysis

VTB.y - Version 1

```
%{
int yylex (void);

void yyerror (char * str)
{
    /* This should handle errors! */
}

%}

%union {
        int number;
        char * string;
        char var;
    }

    %token <number> NUMBER
    %token <string> STRING
    %token <var> VAR_NUM
    %token <var> VAR_STR
    %token LET IF THEN PRINT REM GOTO GOSUB RETURN STOP
           FOR EQUAL TO STEP NEXT SEPARATE
           NEQUAL LT GT LTEQ GTEQ
           ADD SUB MUL DIV
           LPARAN RPARAN

    %start lines

%%
```

AMEY B 50 Amey

Page No.:
Date:

```
lines : line lines
      | /* Nothing */
      ;

line : NUMBER statement
     ;

statement : LET variable EQUAL exp
          | IF test THEN NUMBER
          | PRINT printItem printList
          | REM
          | GOTO NUMBER
          | GOSUB NUMBER
          | RETURN
          | STOP
          | FOR variable EQUAL exp TO exp STEP exp
          | FOR variable EQUAL exp TO exp
          | NEXT variable
          ;

variable : VAR_NUM
         | VAR_STR
         ;

PrintList : SEPARATE printItem
          | /* Nothing */
          ;

PrintItem : exp
          | /* STRING */
          ;
```

AMEY B 50 Amey

Date:

```
test : exp comparison exp
     ;

Comparison : EQUAL
           | NEQUAL
           | LT
           | GT
           | LTEQ
           | GTEQ
           ;

exp : addsub secondary secList
    | secondary secList
    ;

SecList : addsub secondary
        | /* Nothing */
        ;

addsub : ADD
       | SUB
       ;

Secondary : primary priList
          ;

priList : muldiv primary
        | /* Nothing */
        ;

muldiv : MUL
       | DIV
       ;
```

AMEY B 50 Amey

Page No.:
Date:

```
primary : LPARAN exp RPARAN
        | variable
        | NUMBER
        | STRING
        ;
```

Lexical Analysis with Semantics

- In this version of the lexer, the header file generated by yacc/bison is included.
- The header file defines the return values and the union that is used to store the semantic values of tokens. This was created according to the %token declarations and % union part of the grammar file.
  The lexer now extracts values from some types of tokens, and store the values in the yylval union.

VTB.l - version 2

```
%{

  # include "y.tab.h"
  # include <string.h>
  # include <malloc.h>

%}

DIGIT [0-9]
LETTER [A-Z]

%%
```

AMEY B 50 Amey

Page No.:
Date:

```
"LET"          { return LET; }
"IF"           { return IF; }
"THEN"         { return THEN; }
"PRINT"        { return PRINT; }
"REM"[^\n]*    { return REM; }
"GOTO"         { return GOTO; }
"GOSUB"        { return GOSUB; }
"RETURN"       { return RETURN; }
"STOP"         { return STOP; }
"FOR"          { return FOR; }
"TO"           { return TO; }
"STEP"         { return STEP; }
"NEXT"         { return NEXT; }
"\"[^\"]\"     { yylval.string = malloc(strlen(yytext)+1);
                 strcpy(yylval.string, yytext); return STRING; }
"="            { return EQUAL; }
"<>"           { return NEQUAL; }
"<"            { return LT; }
">"            { return GT; }
"<="           { return LTEQ; }
">="           { return GTEQ; }
{LETTER}       { yylval.var = yytext[0]; return VAR-NUM; }
{LETTER}$      { yylval.var = yytext[0]; return VAR-STR; }
{DIGIT}+       { yylval.number = atoi(yytext); return NUMBER; }
","            |
";"            { return SEPARATE; }
"+"            { return ADD; }
"-"            { return SUB; }
"*"            { return MUL; }
"/"            { return DIV; }
"("            { return LPARAN; }
")"            { return RPARAN; }
.              { /* This is an error! */ }

%%
```

AMEY B 50 Amey

Page No.:
Date:

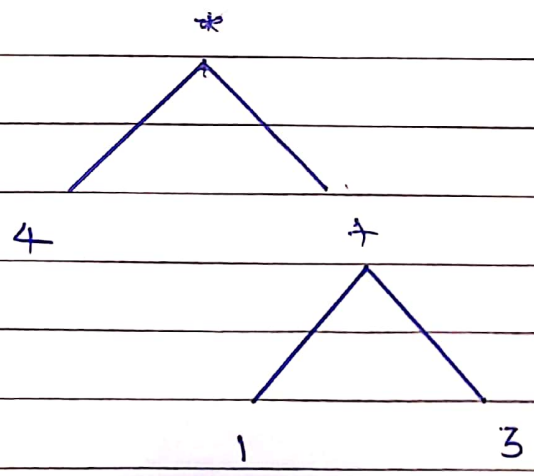
Abstract Syntax Trees

- Abstract syntax trees are an intermediate representation of the code that are created in memory using data structure. The grammatical structure of the language, which has already been defined and has been written down as a YACC grammar file, is translated into a tree structure. Using YACC & C, this means that most grammar rules and tokens become nodes. Comments are an example of what is not put in the tree.

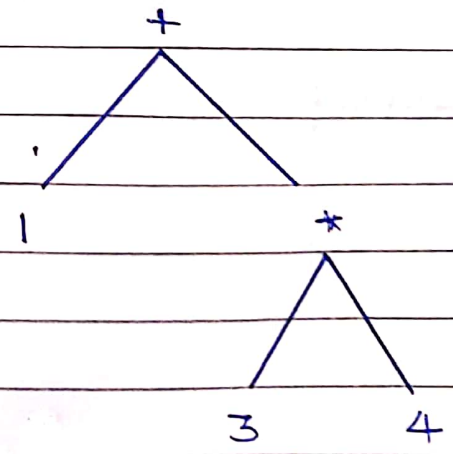
- The grouping of operands is clear within the structure, so tokens such as parentheses do not have to be present in the tree. The same applies to tokens which end blocks of code. This is possible because rules in the grammar file can use these tokens create the tree in the correct shape.

- Illustration of grouping in abstract syntax trees.

(1 + 3) * 4



1 + 3 * 4



AMEY B 50 Amey

Page No.:
Date:

- In this interpreter the primary / secondary expression structures could be discarded by collapsing them. This would add complexity to the code, so it is currently not implemented.
REM statements are also kept (without the comment text) since the definition of VTB implies that they are a valid target for GOTO. In fact, a goto to a non-existent line is undefined, so this interpreter will issue an error.

Generating Abstract Syntax Trees

VTB.y - Version 2

- The lexer is now giving values for tokens and the abstract syntax tree structure has been written. Next the grammar file is updated to construct the trees from what the rules and semantic values. All the tree node types are added to the union declaration. Rules must be given types and return the correct type.