COMPILER-DESIGN LAB

LAB Exercises

Consider the following mini Language, a simple procedural high-level language, only operating on integer data, with a syntax looking vaguely like a simple C crossed with Pascal. The syntax of the language is defined by the following BNF grammar:

```
<program> ::= <block>
<block> ::= { <variabledefinition> <slist> }
        | { <slist> }
<variabledefinition> ::= int <vardeflist> ;
<vardeflist> ::= <vardec> | <vardec> , <vardeflist>
<vardec> ::= <identifier> | <identifier> [ <constant> ]
<slist> ::= <statement> | <statement> ; <slist>
<statement> ::= <assignment> | <ifstatement> | <whilestatement>
            | <block> | <printstatement> | <empty>
<assignment> ::= <identifier> = <expression>
             | <identifier> [ <expression> ] = <expression>
<ifstatement> ::= if <bexpression> then <slist> else <slist> endif
              | if <bexpression> then <slist> endif
<whilestatement> ::= while <bexpression> do <slist> enddo
<printstatement> ::= print ( <expression> )
<expression> ::= <expression> <addingop> <term> | <term> | <addingop>
<term>
<bexpression> ::= <expression> <relop> <expression>
<relop> ::= < | <= | == | >= | > | !=
<addingop> ::= + | -
<term> ::= <term> <multop> <factor> | <factor>
```

<multop> ::= * | /

<factor> ::= <constant> | <identifier> | <identifier> [<expression>]

   | (<expression>)

<constant> ::= <digit> | <digit> <constant>

<identifier> ::= <identifier> <letterordigit> | <letter>

<letterordigit> ::= <letter> | <digit>

<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<empty> has the obvious meaning

Comments (zero or more characters enclosed between the standard C/Java-style comment brackets /*...*/) can be inserted. The language has rudimentary support for 1-dimensional arrays. The declaration int a[3] declares an array of three elements, referenced as a[0], a[1] and a[2]. Note also that you should worry about the scoping of names.

A simple program written in this language is:

```
{ int a[3],t1,t2;
  t1=2;  a[0]=1; a[1]=2; a[t1]=3;
  t2=-(a[2]+t1*6)/(a[2]-t1);
  if t2>5 then
    print(t2);
  else {
    int t3;
    t3=99;
    t2=-25;
    print(-t1+t2*t3);   /* this is a comment   on 2 lines */
  }  endif
}
```

EXERCISE: 1
LEXICAL ANALYZER

Aim:
Design a Lexical analyzer for the above language. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

Input:
A statement or a programming block of above language.

Output:
Symbol table containing information about all the tokens.

Algorithm:
1. Identify the valid tokens i.e., identifiers, keywords, operators etc., in the given language according to above BNF grammar.

2. Write a program for lexical analyzer to recognize all the valid tokens in the input program written according to the grammar.

EXERCISE 2
Lexical Analyzer using LEX

Aim:

Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

Input:
A statement of above language.

Output:
Token type of all the identified tokens in the entered input statement.

Algorithm:
This should contain three distinct parts, each represented by %%

1. Declaration Section:
Any C variable declarations or function prototype required for the actions.

2. Rules Section:
This should consist of a list of regular expression along with their individual actions. Each action is executed only when the regular expression preceding it is recognized by lex.

3. C Functions are defined:
```
main ( )
{
…..
…..
…..
}
```
Lex Translator: lex filename.l
C Compiler: cc lex.yy.c –lfl
Run: ./a.out
More related programs

1. Write a lex program to print the copy of input.

2. Write a lex program to display the number of lines and number of characters in input.
3. Write a lex program to replace the sequence of white spaces by a single blank from input.
4. Write a lex program to replace the sequence of white spaces by a single blank from input text file and add the contents in output text file..

EXERCISE 3
Predictive Parser

Aim:
Design Predictive parser for the given language

Input:
A string w and a parsing table M for grammar G.

Output:
If w is in L(G), a leftmost derivation of w ; otherwise , an error indication.

Algorithm:
Initially, the parser is in a configuration in which it has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

```
set ip to point  to the first symbol of w$;
repeat
   let X be the top stack symbol and a the symbol pointed to by ip;
   if X is a terminal or $ then
          if  X=a then
               pop X from the stack and advance ip.
         Else error()
Else
    If M[X,a]=X->$Y_1Y_2$……..$Y_k$  then begin
               Pop X from the stack;
               Push $Y_k$,$Y_{k-1}$,……..,$Y_1$ onto the stack, with $Y_1$ on top;
   End
   Else  error();
Until  X=$  /*stack is empty*/
```

Exercise 4
YACC

Aim:
Convert the BNF rules into YACC form and write code to generate abstract syntax tree.

Input:
A statement or expression of given language.

Output:
Input statement in the form of abstract syntax tree.

Algorithm:
This contains three sections

```
        Declarations
              %%
        Rules and actions
             %%
        Supporting C- routines
```

Yacc Translator:	**yacc filename.y**
C Compiler:	**cc y.tab.c**
Run:	**./a.out**

EXERCISE 5
LALR PARSER

Aim:
Design LALR bottom up parser for the above language.

Input:
An augmented grammar G'

Output:
The LALR parsing table functions action and goto for G'

Method :

1. Construct C={I_0,I_1,……,I_n}, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core , and replace these sets by their union.
3. Let C'={J_0,J_1,……,J_m) be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i . . If there is a parsing action conflict , the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).
4. The goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is ,J=I_1 U I_2 U ……U I_k , then the cores of goto(I_1,X), goto(I_2,X),….., goto(I_K,X) are the same, since I_1,I_2,….I_K all have the same core as goto(I,X) . Then goto(J,X)=K.

EXERCISE 6
CODE GENERATION

Aim:
To generate machine code from the abstract syntax tree generated by the parser.

Input:
A statement or expression of given language.

Output:
Equivalent code for input statement in the form of given machine readable instructions.

Algorithm:
The following instruction set may be considered as target code.
The following is a simple register-based machine, supporting a total of 17 instructions. It has three distinct internal storage areas. The first is the set of 8 registers, used by the individual instructions as detailed below, the second is an area used for the storage of variables and the third is an area used for the storage of program. The instructions can be preceded by a label. This consists of an integer in the range 1 to 9999 and the label is followed by a colon to separate it from the rest of the instruction. The numerical label can be used as the argument to a jump instruction, as detailed below. In the description of the individual instructions below, instruction argument types are specified as follows:

R	specifies a register in the form R0, R1, R2, R3, R4, R5, R6 or R7 (or r0, r1, etc.).
L	Specifies a numerical label (in the range 1 to 9999).
V	Specifies a ``variable location'' (a variable number, or a variable location pointed to by a register - see below).
A	Specifies a constant value, a variable location, a register or a variable location pointed to by a register (an indirect address). Constant values are specified as an integer value, optionally preceded by a minus sign, preceded by a # symbol. An indirect address is specified by an @ followed by a register. So, for example, an A-type argument could

have the form 4 (variable number 4), #4 (the constant value 4), r4 (register 4) or @r4 (the contents of register 4 identifies the variable location to be accessed).

The instruction set is defined as follows:

LOAD A,R
loads the integer value specified by A into register R.

STORE R,V
stores the value in register R to variable V.

OUT R
outputs the value in register R.

NEG R
negates the value in register R.

ADD A,R
adds the value specified by A to register R, leaving the result in register R.

SUB A,R
subtracts the value specified by A from register R, leaving the result in register R.

MUL A,R
multiplies the value specified by A by register R, leaving the result in register R.

DIV A,R
divides register R by the value specified by A, leaving the result in register R.

JMP L
causes an unconditional jump to the instruction with the label L.

JEQ R,L
jumps to the instruction with the label L if the value in register R is zero.

JNE R,L

jumps to the instruction with the label L if the value in register R is not zero.

JGE R,L

jumps to the instruction with the label L if the value in register R is greater than or equal to zero.

JGT R,L

jumps to the instruction with the label L if the value in register R is greater than zero.

JLE R,L

jumps to the instruction with the label L if the value in register R is less than or equal to zero.

JLT R,L

jumps to the instruction with the label L if the value in register R is less than zero.

NOP

is an instruction with no effect. It can be tagged by a label.

STOP

stops execution of the machine. All programs should terminate by executing a STOP instruction.