COL 226: Assignment 2

1 Introduction

In this assignment, you have to implement a parser for the language PL0. For this, you have to read the input tokens from a text file, which was the output of the scanner. For this assignment, you have to implement a Recursive Descent parser, which is a top-down parser.

2 Output of the Program

The output of the program would be written into two files:

- The first file will contain the parse tree generated by the parser.
- The second file will contain the symbol table.

Recursive descent parser generates a tree strcuture which needs to be written to a text file using a toString function given in the appendix. In order to use the toString function, define the node of tree as given in the appendix. The tree will be stored using matched paranthesis. Consider the following piece of code in PL0:

```
int a, b;
{
        a := 5;
        b := a+4;
}
```

The scanner output for the above code would be:

```
INT(1,1)
IDENT(1,5,a)
COMMA(1,6)
IDENT(1,8,b)
EOS(1,9)
LB(2,1)
IDENT(3,2,a)
ASSIGN(3,4)
INTLITERAL(3,6,5)
EOS(3,7)
IDENT(4,2,b)
ASSIGN(4,4)
IDENT(4,6,a)
BINADD(4,7)
INTLIT(4,8,4)
EOS(4,9)
RB(5,1)
```

The scanner output would be input to your program. The graphic version of the parse tree is given as "Parsetree.html". This parse tree should be written in text form using nested Square Brackets. All the nodes at the same level should be emcompassed in a single bracket and separated by comma. The resultant parse tree for the above program is given below. Do not worry about whitespace. You can use tabs to indent or write the entire tree in a single line. However, proper spacing and indentation will improve the readibility of the output(helpful during demo). The program would be autograded and the comparison would be made after stripping the program of all the white space. All the non-terminals would be written according to the

notations given in the grammar and all the tokens will have the ascii representation. For identifiers, we will use the names of the identifier and each of these identifier should have an entry in the Symbol table.

```
[ Program
 [ Block
  [DeclarationSeq
   [ varDecls
    [ INT, Ident[a], COMMA, Ident[b], EOS
    ]
   ],
   CommandSeq
   [ LB, Command
    [ AssignmentCmd
     [Ident [a], ASSIGN, Expression [IntExpression [ IntLiteral [5] ] ] ]
     ],
   EOS, Command
    [ AssignmentCmd
     [ Ident [b], ASSIGN, Expression
      [ IntExpression
       [AddExpression
        [IntExpression[Ident[a]], BINADD, IntExpression[ IntLiteral [ 5 ] ] ]
       ]
      ]
     ]
    ], EOS
   ]
  ]
 ]
]
```

In addition to the parse tree, you have to output a symbol table as well. The first n rows of the symbol table will contain keywords. The rest of the rows will contain Indentifiers with their values. The structure and toString function for the symbol table is given in **Appendix A**.

3 The Language PL0

3.1 Tokens of Language PL0

Reserved words. int, bool, tt, ff, if, then, else, while, proc, print, read.

Integer Operators.

Unary. ~

Binary. +, -, *, /, %

Boolean Operators.

Unary. !

Binary. &&, ||

Relational Operators. =, <>, <, <=, >, >=

Assignment. :=

Brackets. (,), {, }

Punctuation. ;, ,

Identifiers. (A-Za-z) (A-Za-z0-9)*

Comment. Any string of printable characters enclosed in (*, *)

3.2 EBNF of Language PL0

Program ::= *Block* .
Block ::= *DeclarationSeq CommandSeq* .
DeclarationSeq ::= *[VarDecls] [ProcDecls]* .
VarDecls ::= *[IntVarDecls] [BoolVarDecls]* .
IntVarDecls ::= int *Ident {, Ident};* .
BoolVarDecls ::= bool *Ident {, Ident};* .
ProcDecls ::= *[ProcDef {;ProcDecls};]* .
ProcDef ::= proc *Ident Block* .
CommandSeq ::= *{{Command;}}*.
Command ::= *AssignmentCmd* | *CallCmd* | *ReadCmd* | *PrintCmd* | *ConditionalCmd* | *WhileCmd* .
AssignmentCmd ::= *Ident* := *Expression* .
CallCmd ::= *Ident* .
ReadCmd ::= read*(Ident)* .
PrintCmd ::=print*(Ident)* .
Expression ::= *IntExpression* | *BoolExpression* .
ConditionalCmd ::= if *BoolExpression* then *CommandSeq* else *CommandSeq* .
WhileCmd ::= while *BoolExpression CommandSeq* .

3.3 ASCII representation of tokens

The representation of tokens (in ASCII) to be produced at the end of scanning for text-file output is as follows:

Token	**ASCII representation**
+	"BINADD"
~	"UNMINUS"
-	"BINSUB"
/	"BINDIV"
*	"BINMUL"
%	"BINMOD"
!	"NEG"
&&	"AND"
||	"OR"
:=	"ASSIGN"
=	"EQ"
<>	"NE"
<	"LT"
>	"GT"
<=	"LTE"
>=	"GTE"
(	"LP"
)	"RP"
{	"LB"
}	"RB"
;	"EOS"
,	"COMMA"
int	"INT"
bool	"BOOL"
if	"IF"
then	"THEN"
else	"ELSE"
while	"WHILE"
proc	"PROC"
print	"PRINT"
read	"READ"
(A-Za-z)(A-Za-z0-9)*	"IDENT"

(0-9)(0-9)*	"INTLIT"
tt,ff	"BOOLVAL"

3.4 Structure of Nodes

The nodes of the parse tree can be broadly divided into Non-Terminals and Terminals. For the Non-terminals, use the representation given in the grammar while for Terminals or tokens, use the ASCII representation. Here's an explanation of the various node types provided:

NONTERMINAL of string*int First field contains the name of the non-terminal and second field is the level of the node.

INTLITERAL of int*int First field represents the value of the literal and second field is the level of the node.

BOOLLITERAL of string*int First field is 'tt' or 'ff' and the second field is the level of the node.

ERROR of int * string * int First field is the row number in which the error occured and second field correspond to the statement in which the error occured.

3.5 Structure of Symbols

The structure of symbols (of Symbol Table) to be produced is given in SML, in **Appendix A**.

Explanation for the structure of each symbol is given below:

KEYWORD of string * string: First field, string, is the keyword itself and the second field is the token representation of the keyword.

INTSYMBOL of string: First field is the name of identifier.

BOOLSYMBOL of string: First field is the name of the idetifier.

PROCSYMBOL of string: The field is the name of the identifier.

When you are printing the symbols, the toString function add the type of the identifier to the entry. For example, INT would ve added for an Identifier of type integer.

3.6 Note:

- Infix notations for binary operators and preorder notation for unary operator is required.
- Relational operators are for both booleans and integers, and ff $<$ tt (means ff is smaller than tt).
- You will be defining "fromString" function to read from file as a string and process accordingly.
- Language is case-sensitive.
- You can code either in SML or OCaml or Haskell, but imperative paradigm such as "for" loops, "while" loops, etc. are strictly prohibited.
- Your assignment will be autograded first. So make sure that you submit the assignment in given format and output is printed to file in the predefined format.
- For the sake of running test cases,the program file should accept two arguments, the first argument being the location of the input file and second being the location of the output file.

4 Instructions for Submission

- **All submissions must be through moodle. No other form of submission will be entertained.**
- **No submissions will be entertained after the submission portal closes.**
- **Sometimes there are two deadlines possible – the early submission deadline (which we may call the "lifeline") and the final "deadline". All submissions between the "lifeline" and the "deadline" will suffer a penalty to be determined appropriately.**

5 What to Submit?

- You will create one folder which will have 2 files, program file and the writeup file.
- The program file should be named with your Kerberos ID. For example, if kerberos id is 'cs1140999' then the file name should be cs1140999.sml or cs1140999.ocaml. The writeup should be named as "writeup.txt".
- Both the files should be present in one folder. Your folder also should be named as your Kerberos ID. For example, if kerberos id is 'cs1140999' then the folder should be called cs1140999.
- The first line of writeup should contain a numeral indicating language preferred with 0-ocaml, 1-sml and 2-haskell.
- For submission, the folder containing the files should be zipped(".zip" format). Note that, you have to zip folder and NOT the files.
- This zip file also should have name as your Kerberos ID. For example, if kerberos id is 'cs1140999' then the zip file should be called cs1140999.zip.
- Since the folder has to be zipped the file cs1140999.zip should actually produce a new folder cs1140999 with files (cs1140999.sml or cs1140999.ocaml) and writeup.txt.

  Hence the command *"unzip -l cs1140999.zip"* should show

  cs1140999/cs1140999.sml
  cs1140999/writeup.txt
- After creating zip, you have to convert ".zip" to base64(.b64) format as follows (for example in ubuntu): *base64 cs1140999.zip > cs1140999.zip.b64* will convert .zip to .zip.b64
  This cs1140999.zip.b64 needs to be uploaded on moodle.
- After uploading, please check your submission is up-to the mark or not, by clicking on evaluate. It will show result of evaluation. If folder is as required, there will be no error, else REJECTED with reason will be shown. So, make sure that submission is not rejected.

Appendix A

```
Datatype NODE = NONTERMINAL of string*int
| INTLITERAL of int * int
| BOOLLITERAL of string * int
| ERROR of int * string * int

Datatype SYMBOL = KEYWORD of string * string
| INTSYMBOL of string
| BOOLSYMBOL of string
| PROCSYMBOL of string

fun toString NODE =
case NODE of
NONTERMINAL(a, b) => a
| INTLITERAL(a, b) => Int.toString a
| BOOLLITERAL(a, b) => a
| ERROR(a, b, c) => "ERROR in line " ^ Int.toString a ^ ": " ^ b ""

fun toString SYMBOL =
case SYMBOL of
KEYWORD(a,b) => a ^"\t" ^b ^"\n "
| INTSYMBOL(a,b) =>  a ^"\tINT\n "
| BOOLSYMBOL(a,b) => a ^"\tBOOL\n "
| PROCSYMBOL(a) => a ^"\tPROC\n "
```