COL 226: Assignment 2

1 Introduction

In this assignment, you have to implement a parser for the language PL0 3. 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.

NOTE: The assigment has been updated to include LL(1) form of the language. A new parse tree has been created for the same which is included with this assignment. The assignment will be autograded. So, please ensure that your output is in the correct format.

2.1 Parse Tree

Recursive descent parser generates a tree structure which needs to be written to a text file using a toString function. The tree will be stored using matched paranthesis. Consider the following piece of code in PL0:

```
int a;
{
}
```

The scanner output for the above code would be:

```
INT(1,1)
IDENT(1,5,a)
EOS(1,6)
LB(2,1)
RB(3,1)
```

The scanner output (given above) will be input to your program for this assignment. You have to write a fromString function to read the above file. The parse tree generated for the above has been added as the "TestCase1/Output_ParseTree.txt" file. You can use that you compare your output. 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. 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 in case there is a demo). All the non-terminals are to be written according to the notations given in the grammar and all the tokens according to the ASCII representation. For identifiers, you have to use the names of the identifier and each of these identifier should have an entry in the symbol table 2.2.

2.2 Symbol Table

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 all the keywords of language PL0. The rest of the rows will contain identifiers (of input program) with their types. The structure and toString function for the symbol table is given in **Appendix A**. The symbol table for the example program is as follows:

int	INT		
bool	BOOL		
tt	BOOLVAL		
ff	BOOLVAL		
if	IF		
then	THEN		
else	ELSE		
while	WHILE		
proc	PROC		
print	PRINT		
read	READ		
call	CALL		
a	IDENT	INT	global

3 The Language PL0

3.1 Tokens of Language PL0

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

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 (*, *)

4 Language PL0 in ll(1) form

Program ::= *Block* .
Block ::= *DeclarationSeq CommandSeq* .
DeclarationSeq ::= *VarDecls ProcDecls* .
VarDecls ::= *IntVarDecls BoolVarDecls* .
IntVarDecls ::= int *VarDef* | ϵ .
BoolVarDecls ::= bool *VarDef* | ϵ .
VarDef ::= *Ident VarDef1* .
VarDef1 ::= ,*VarDef* | ; .
ProcDecls ::= proc *Ident Block* ; *ProcDecls* | ϵ .
CommandSeq ::= { *Command* }.
Command ::= *AssignmentCmd* ; *Command*

| *CallCmd* ; *Command*
| *ReadCmd* ; *Command*
| *PrintCmd* ; *Command*
| *ConditionalCmd* ; *Command*
| *WhileCmd* ; *Command*
| ϵ .

AssignmentCmd ::= *Ident* := *Expression* .
CallCmd ::= call *Ident* .
ReadCmd ::= read(*Ident*) .
PrintCmd ::=print(*Ident*) .
ConditionalCmd ::= if *BoolExpression* then *CommandSeq* else *CommandSeq* .
WhileCmd ::= while *BoolExpression CommandSeq* .
Expression ::= *BoolExpression* .
IntExpression ::= *IntT IntE*.
IntE ::= +*IntExpression* | − *IntExpression* | ϵ.
IntT ::= *IntF IntT1*.
IntT1 ::= % *IntT* | ∗ *IntT* | / *IntT* | ϵ.
IntF ::= ∽*IntF1* | *IntF1*.
IntF1 ::= *Ident* | *IntLiteral* | (*BoolExpression*).
BoolExpression ::= *BoolF BoolE* .
BoolE ::= || *BoolExpression* | ϵ .
BoolF ::= *BoolG BoolF1* .
BoolF1 ::= && *BoolF* | ϵ .
BoolG ::= *BoolH BoolG1* .
BoolG1 ::= = *BoolG* | <> *BoolG* | ϵ .
BoolH ::= *BoolI BoolH1* .
BoolH1 ::= < *BoolH* | <= *BoolH* | > *BoolH* | >= *BoolH* | ϵ .
BoolI ::= !*BoolJ* | *BoolJ* .
BoolJ ::= *BoolLiteral* | *IntExpression* .

4.1 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"
call	"CALL"

4.2 Structure of Nodes of the Parse Tree

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. Terminals can be of various types: Keywords or Tokens wil be written using their ASCII representation. E.g. "if" will be written as "IF", the token "," will be written as "COMMA". The IntLiteral will be an integer. The BoolLiteral would be a string : "tt" or "ff". For Identifiers, the name of the identifier will be printed. The error node of the parse tree should print "ERROR at ("row no", "column no")". For example, for the statement:

a := b#4;

Let the statement be in row 5 of the program. Your program should output: "ERROR at (5, 8)". Do not worry about printing the statement. Also, note that some terminals are empty strings. For these, print out "EPSILON" as shown in the parse tree and has the same variable names.

4.3 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 * string: First field is the name of identifier and second field is it's scope.

BOOLSYMBOL of string * string: First field is the name of the idetifier and second field is it's scope.

PROCSYMBOL of string* string: The field is the name of the identifier and second field is it's scope.

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.

4.4 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.
- The language is right-associative.

- For the sake of running test cases,the program file should accept three arguments, the first argument being the location of the input file and second argument being the location of the output file for parse tree and the third argument being the location of the output file for symbol table. For example: cs1140999.sml input.txt output_parseTree.txt output_symbolTable.txt

5 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.**

6 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 SYMBOL = KEYWORD of string * string
| INTSYMBOL of string * string
| BOOLSYMBOL of string * string
| PROCSYMBOL of string * string
```

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