Assignment 1

1 Introduction

The EBNF specification of a programming language is a collection of rules that defines the (context-free) grammar of the language. It specifies the formation rules for the correct grammatical construction of the phrases of the language.

Start symbol. The rules are written usually in a "top-down fashion" and the very first rule gives the productions of the start symbol of the grammar.

Non-terminals. Uses English words or phrases to denote non-terminal symbols. These words or phrases are suggestive of the nature or meaning of the constructs.

Metasymbols.

- Sequences of constructs enclosed in "{" and "}" denote zero or more occurrences of the construct (c.f. Kleene closure on regular expressions).
- Sequences of constructs enclosed in "[" and "]" denote that the enclosed constructs are optional i.e. there can be only zero or one occurrence of the sequence.
- Constructs are enclosed in "(" and ")" to group them together.
- "| " separates alternatives.
- "::= " defines the productions of each non-terminal symbol.
- " ." terminates the possibly many rewrite rules for a non-terminal.

Terminals. Terminal symbols strings are usually enclosed in double-quotes when written in monochrome (we shall additionally code them in Green).

2 Language PL0

You are given EBNF of one such language *"PL0"*. In this course, you will be designing a compiler for this very simple Programming Language, piece-wise, guided with a series of assignments. For this assignment, you have to *individually* program a scanner which outputs tokens in some printable form into a text file.

The scanner, is usually based on a finite-state machine (FSM), which reads character by character from the file and detects tokens present in it. For instance, an integer token may contain any sequence of numerical digit characters.

Your scanner should be able to read a text file as input, find out all the tokens and output it to another text file.

2.1 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 (*, *)

2.2 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 r::= {{Command;}}.
Command r::= AssignmentCmd | CallCmd | ReadCmd | PrintCmd | ConditionalCmd | WhileCmd .
AssignmentCmd ::= Ident := Expression .
CallCmd r::= Ident .
ReadCmd ::= read*(Ident) .*
*PrintCmd ::=*print*(Ident) .*
Expression ::= IntExpression | BoolExpression .
ConditionalCmd ::= if *BoolExpression* then *CommandSeq* else *CommandSeq .*
WhileCmd ::= while *BoolExpression CommandSeq .*

2.3 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"

2.4 Structure of tokens

The structure of tokens (as records) to be produced at the end of scanning is given in SML, in **Appendix A**.

Explanation for the structure of each token type is given below:

INTLIT of int * int * int: First field, int, is for row number, second field, int, is for column number from which literal is starting and third field, int is representing integer value of that literal.

IDENT of int * int * string: First two fields are same as above, third field, string represents identifier itself.

BOOLVAL of int * int * bool: First two fields are same as above, third field, bool represents Boolean value, either true (tt in given grammar) or false (ff in given grammar).

ERROR of int * int * int: First two fields are same as above, third field, int represents size of the lexeme, which is invalid token.

For rest of the tokens, first field, int, is for row number, second field, int, is for column number. Special token, *"ERROR"* is introduced in the structure of token, which is generated if we cannot find any token which matches with the tokens specified for this language. In that case, token *"ERROR"* should be generated and scanning for more tokens should be continued (proceed further even if you get *"ERROR"* token).

You are also given *toString* function for tokens, which converts given token into its string representation. You need to use this function to print output to file. So, you have to define one more *toString* function, which uses *toString* function given to you and prints output to a file. Note that, while printing output to file, each token should be in a separate line. *toString* function, which is provided to you, takes token as parameter and gives its string representation and this string representation is followed by new line character. So, you need NOT to add new line character explicitly.

2.5 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.

3 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.**

4 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

Appendix A

```
datatype TOKEN = UNMINUS of int * int
               | BINADD of int * int
               | BINSUB of int * int
               | BINDIV of int * int
               | BINMUL of int * int
               | BINMOD of int * int
               | NEG of int * int
               | AND  of int * int
               | OR  of int * int
               | ASSIGN of int * int
               | EQ of int * int
               | NE of int * int
               | LT of int * int
               | LTE of int * int
               | GT of int * int
               | GTE of int * int
```

```
                | LP  of int * int
                | RP of int * int
                | LB of int * int
                | RB of int * int
                | EOS of int * int
                | COMMA of int * int
                | INT of int * int
                | BOOL of int * int
                | IF of int * int
                | THEN of int * int
                | ELSE of int * int
                | WHILE of int * int
                | PROC of int * int
                | PRINT of int * int
                | READ of int * int
                | ERROR of int * int * int
                | INTLIT of int * int * int
                | IDENT of int * int * string
                | BOOLVAL of int * int * bool


fun toString TOKEN =
    case TOKEN of
        UNMINUS(a,b)    =>       "UNMINUS(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BINADD(a,b)    =>       "BINADD(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BINSUB(a,b)    =>       "BINSUB(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BINDIV(a,b)    =>       "BINDIV(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BINMUL(a,b)    =>       "BINMUL(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BINMOD(a,b)    =>       "BINMOD(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | NEG(a,b)    =>       "NEG(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | AND(a,b)    =>       "AND(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | OR(a,b)    =>       "OR(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | ASSIGN(a,b)    =>       "ASSIGN(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | EQ(a,b)    =>       "EQ(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | NE(a,b)    =>       "NE(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | LT(a,b)    =>       "LT(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | LTE(a,b)    =>       "LTE(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | GT(a,b)    =>       "GT(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | GTE(a,b)    =>       "GTE(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | LP(a,b)    =>       "LP(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | RP(a,b)    =>       "RP(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | LB(a,b)    =>       "LB(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | RB(a,b)    =>       "RB(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | EOS(a,b)    =>       "EOS(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | COMMA(a,b)    =>       "COMMA(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | INT(a,b)    =>       "INT(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | BOOL(a,b)    =>       "BOOL(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | IF(a,b)    =>       "IF(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | THEN(a,b)    =>       "THEN(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | ELSE(a,b)    =>       "ELSE(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | WHILE(a,b)    =>       "WHILE(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | PROC(a,b)    =>       "PROC(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | PRINT(a,b)    =>       "PRINT(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | READ(a,b)    =>       "READ(" ^ Int.toString a ^ "," ^ Int.toString b ^ ")\n"
      | ERROR(a,b,c)=>"ERROR(" ^ Int.toString a ^ "," ^ Int.toString b ^ "," ^ Int.toString c ^")\n"
      | INTLIT(a,b,c)=>"INTLIT(" ^ Int.toString a ^ "," ^ Int.toString b ^ "," ^ Int.toString c ^")\n"
      | IDENT(a,b,c)=>"IDENT(" ^ Int.toString a ^ "," ^ Int.toString b ^ "," ^ c ^ ")\n"
      | BOOLVAL(a,b,c)=>"BOOLVAL("^ Int.toString a ^ "," ^ Int.toString b ^ "," ^ Bool.toString c ^")\n"
```