CS315-Programming Languages

Project 01 Report

Language Name: Logos

Aldo Tali (21500097 – Section 3)

Faaiz Ul Haque (21503527 – Section 1)

Unas Sikander Butt (21503511 – Section 2)

1.0 Terminal Statement Definitions

The following file is a lex file which will define all the terminals that will be used by our language on parsing time. These include identifiers for out variables, constants, integers and characters/words that will act as keywords in the building and the syntax of the language as a whole. All these will be used on the BNF to simplify the expressions and the relations of the statements that the language supports. Note that these texts are in purple. Throughout this report this color will be used to imply/identify that the use of a particular statement leads to a predefined terminal which can be found in these definitions in lex.

```
%option main
LP              \(
RP              \)
LCBRACE         \{
RCBRACE         \}
LSBRACKET       \[
RSBRACKET       \]
COMMA           \,
COMMENT         \/\/[^\n]*
TRUE            true
FALSE           false
AND             \/\\
OR              \\\/
IMPLIES         \-\>
NEGATION        \~
ADD_OP          \+
SUB_OP          \-
EQUALS          \=\=
MULT_OP         \*
DIV_OP          \/
MOD_OP          \%
GREATER         \>
LESS            \<
GREATER_EQ      \>\=
LESS_EQ         \<\=
NOT_EQ          \~\=
LETTER          [A-Za-z]
ASSIGN          \=
GIVE            give
TAKE            take
IF              if
THEN            then
```

```
ELSE            else
LOOP            loop
ENDLOOP         endloop
FUNCTION        function
STRING          \"(\\.|[^"\\])*\"
DIGIT           [0-9]
BOOLEAN         {TRUE}|{FALSE}
POSITIVE_D      \(\+{DIGIT}+\)
NEGATIVE_D      \(\-{DIGIT}+\)
INT             {POSITIVE_D}|{NEGATIVE_D}|{DIGIT}+
RAT             {INT}\.[0-9]+|\.{INT}
CONSTANT        {STRING}|{INT}|{RAT}|{BOOLEAN}
IDENTIFIER      {LETTER}+(\_|\-|{DIGIT}|{LETTER})*
LOGIC_OP        {AND}|{OR}|{IMPLIES}|{NEGATION}
COMP_OP
{GREATER}|{GREATER_EQ}|{LESS}|{LESS_EQ}|{EQUALS}|{NOT_EQ}
ALGEB_OP        {ADD_OP}|{SUB_OP}|{MULT_OP}|{DIV_OP}|{MOD_OP}
```

2.0 BNF Description

The following gives the constructs of our language by using the BNF grammar rules. Note that in this section all the constructs are build in red color. When the BNF statements reduce to a terminal, the definition can be looked up in the previous section by locating that terminal in the lex file. Each language specifics is followed by a BNF structure and a short comment of what this definition includes and what it does not include. The following structures have been proposed based on the structures of other languages such as Java, C, C++,Python and also based on the course textbook.

<BOOLEAN> **=>** <TRUE> | <FALSE>

BOOLEAN holds the truth values available in the proposed language. These values can only be two, true or false. These are commonly used in conditional, control statements and in logical operations of prepositional calculus.

<CONSTANT> **=>** <STRING> | <INT> | <RAT> | <BOOLEAN>

CONSTANT is defined in the language as a definition which can be a string, integer, boolean value or a rational value. In our case these are simplified into STRING, INT, BOOLEAN, RAT.
A string will hold all characters between the two boundary quotation marks. If a user wants to use a quotation mark in the string (") the user has to use the escape character (\). For example for the input: "hello \"CS315\" world" the output would be: hello "CS315" world.
An integer can be either a positive integer (+5) or a negative integer (-5) or simply a normal integer (combination of one or more digits with one another).
In our language syntax positive and negative numbers have to be enclosed in parentheses. (example: a + (-5) + (+5) + 2)
A rational number includes numbers that start with a digit and after combining one or more of these they are followed by a dot and a combination of some other digits. The language also recognizes the rational numbers as a dot followed by a number which means that the number is assumed to be zero dot a fractional part. (example: .02, 5.3, (+2.7))

<IDENTIFIER> => <LETTER><IDENTCHARS>

IDENTIFIER holds all variables. These variables are restricted to starting with a letter and then they can be followed by letters, underscores, hyphens or digits. A variable in our language is a structure which will be able to keep all the values that we want to store for later use.

<IDENTCHARS> **=>** <LETTER><IDENTCHARS>

| <DIGIT><IDENTCHARS>
| <UNDERSCORE><IDENTCHARS>
| <HYPHEN><IDENTCHARS>
| ε

IDENTCHARS are all possible characters used in identifiers. These include -, _, alphabets and digits. The structure is a helper to define the Identifiers in the languages for us to resolve the possible ambiguities.

<LOGIC_OP> **=>** <AND> | <OR> | <IMPLIES>

LOGIC_OP holds all prepositional calculus operators that are defined in our language except for the negation operator. The reason for this is that a negation operator comes always in front of a statement whereas these operators always appear in between two statements.

<LOGIC_STMT> **=>** <PREDIFINED_LOGIC> | <COMPLEX_LOGIC>

LOGIC_STMT can simply be PREDFINED_LOGIC (i.e, constant, boolean, identifier) or COMPLEX_LOGIC (i.e a/\b, 5\/c, x->y, ~a). The division here into two separate logic structures is a design choice that facilitates the writability of the programs and avoids the possible ambiguity when trying to fetch the correct statement when parsing.

<PREDEFINED_LOGIC> **=>** <BOOLEAN> | <CONSTANT> | <IDENTIFIER>

PREDEFINED_LOGIC can be a constant, a boolean value or an identifier (a variable). Since these are all simple types when trying to compare and their values can be easily determined and be predefined in the yacc implementation in the second part of our project, we have chosen to put then into a separate structure in the BNF.

<COMPLEX_LOGIC> **=>** (LOGIC_STMT)
| <NEGATION> <LOGIC_STMT>
| <LOGIC_STMT> <LOGIC_OP> <LOGIC_STMT>

COMPLEX_LOGIC is a structure to define how to parse the more complex prepositional statements of the language. The first rule will identify a logic statement that is found inside two round brackets. The second rule will catch the negation of any logical statement; let it be simple or complex. If the input does not fall in any of the first two categories then the statement will just be divided into two smaller statements separated in between with a logical operator.

<LOOP_STMT> **=>** <LOOP>(<CONDITION>)<STATEMENTS><ENDLOOP>

LOOP_STMT is used to repeat statements dependent on given conditions. The structure is defined by the keyword loop followed by a curly open brace, a condition to evaluate, closing curly brace, and some statements that execute in the

loop block followed in the end by the keyword endloop which signifies the end of the loop scope.

<IF_STMT> => <IF_ONLY> | <IF_ELSE>

IF_STMT is either IF_ONLY or IF_ELSE. This means that the language will only deal with if statements which have a single condition or a double condition case.

<IF_ELSE> => <IF>(<CONDITION>)<THEN><IF_ELSE><ELSE><IF_ELSE>
| <STATMENTS>

IF_ELSE is an IF_STMT which is followed by an else which deals with the second case of the condition.

<IF_ONLY> => <IF>(<CONDITION>)<THEN><STATEMENTS>
| <IF>(<CONDITION>)<THEN><IF_ELSE><ELSE><IF_ONLY>

IF_ONLY is an IF_STMT without a matching else.

<CONDITION> => <LOGIC_STMT> | <COMPARATIVE_STMT>

CONDITION is either a logic statement which results in a boolean, or it can be a COMPARATIVE_STMT which is defined below.

<COMPARATIVE_STMT> => <COMPARATORS> <COMP_OP> <COMPARATORS>

COMPARATIVE_STMTS are statements which compare two COMPARATORS which are defined below.

<COMP_OP> => <GREATER> | <GREATER_EQ> | <LESS> | <LESS_EQ>
| <EQUALS> | <NOT_EQ>

COMP_OP holds all comparative operators for a particular statement.

<COMPARATORS> => (<ALGEB_STMT>) | <ALGEB_LIST>

COMPARATORS can either be an algebraic statements or a list of algebraic variables or constants which are defined below.

<ALGEB_STMT> => <ALGEB_LIST><ALGEB_OP>(<ALGEB_STMT>)
|<ALGEB_LIST><ALGEB_OP><ALGEB_LIST>
| (<ALGEB_STMT>)

ALGEB_STMTS holds all algebraic statements.

<ALGEB_LIST> => <IDENTIFIER> | <CONSTANT>

ALGEB_LIST are variables or constants which are used in algebraic expressions.

<ASSIGN_STMT> => <IDENTIFIER><ASSIGN><STMT_LIST>

ASSIGN_STMT is used to assign a value to an identifier using STMT_LIST defined below.

<STMT_LIST> => <COMPARATIVE_STMT>
| <ALGEB_STMT>

| <LOGIC_STMT>
| <ASSIGN_STMT>
| <IF_STMT>
| <LOOP_STMT>

STMT_LIST is a collection of the statements defined above. It is used to define these statements in a recursive manner.

<INPUT_STMT> **=>** <TAKE><IDENTIFIER>

INPUT_STMT is used to take an input and store it in an identifier.

<OUTPUT_STMT> **=>** <GIVE><PREDEFINED_LOGIC>

OUTPUT_STMT is used to output. The keyword give can be followed by either a identifier or a constant.

<STATEMENTS> **=>** <STMT_LIST>
| <STMT_LIST><STATEMENTS>
| <INPUT_STMT>
| <OUTPUT_STMT>
| <INPUT_STMT><STATEMENTS>
| <OUTPUT_STMT><STATEMENTS>

STATEMENTS combines the INPUT_STMT, OUTPUT_STMT and the STMT_LIST in all possible combinations.

<PREDICAMENT> **=>** <FUNCTION><WHITE_SPACE><IDENTIFIER> (<PARAM_LIST>){ <STATEMENTS> }

PREDICAMENT is used to declare a user defined function. The syntax uses the keyword function followed by a white space and an identifier which is the function name, a bracket which opens the parameter list input mode, a closing right parenthesis for the parameter lists and last a number of statements inside curly braces.

<PARAM_LIST> **=>** <IDENTIFIER><COMMA><PARAM_LIST>
|<IDENTIFIER>

PARAM_LIST contains all the possible parameters of the function. These can be identifiers separated by comma's in between.

<ARGS_LIST> **=>** <IDENTIFIER><COMMA><ARGS_LIST>
|<CONSTANT><COMMA><ARGS_LIST>
|<IDENTIFIER>
|<CONSTANT>

ARGS_LIST contains all the possible arguments of the function.

<FUNCTION_CALL> **=>** <IDENTIFIER> (<ARGS_LIST>);

FUNCTION_CALL will be used to instantiate a pre-defined function. It can take either variables or constants as parameters.

3.0 Motivations and Constraints

Readability & Realiability & Writability:

Comments: The syntax of the comments has been kept simple and follows other language conventions, such as java and c++. The comment symbol (//) is written at the start of the line which helps the readability of the program in distinguishing between comments and code. The user must write the token on every new line allowing for a reliable program as the code is less prone to mistakes. As this is not a complex token the writability of the program is straight forward.
Identifiers: Again the common code convention of having variable names starting with letters is used in our identifiers. Complex symbols are not allowed in our variables. The only characters that are use-able are alphabets, digits, -, and _. This assists the writability and readability of the program. Since the identifiers do not allow complex characters, the program is less likely to make mistakes resulting in a more reliable language. Additionally this makes the debugging process more convenient.
Reserved Words: function, true,false, give, take, if, then, else, loop,endloop.
Our reserved words are kept short and simple with meaningful names to give users an easy to implement syntax. This also makes understanding the code by the programmers easier, supporting readability of the program. Again in this case, the code becomes reliable as there will be less mistakes.