**Bilkent University**

**CS-315**

**Project Part 1**

**Language Name: Syntax**

A Language By:

Unas Sikander Butt (21503511 – Section 2)

Faaiz Ul Haque (21503527 – Section 1)

Aldo Tali (21500097 – Section 3)

**Complete BNF Description**

Note: All texts in purple are defined in our lex file.

Our constructs are given in red.

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

**BOOLEAN** holds either true or false values. It is commonly used in conditional and control statements.

We further simplified it in Lex as follows:

TRUE **=>** true

FALSE **=>** false

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

**CONSTANT** can be a string, int, boolean or RAT. In this case, RAT is defined as a double.

We further simplified it in Lex as follows:

STRING **=>** \”(\\.|[^”\\])\*\”

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 backslash character (\). For example input: “hello \”CS315\” world” output: hello ”CS315” world.

INT **=>** {POSITIVE\_D} | {NEGATIVE\_D} | {DIGIT}+

An INT can be either a positive digit (+5) or a negative digit (-5) or simply a digit (5).

POSITIVE\_D **=>** (\+{DIGIT}+)

NEGATIVE\_D **=>** (\-{DIGIT}+)

DIGIT **=>** [0-9]

In our language syntax positive and negative numbers have to be enclosed in parentheses. (example: a + (-5) + (+5) + 2)

RAT **=>** {INT}\.[0-9]+|\.{INT}

A RAT includes all rational numbers. (example: .02, 5.3, (+2.7))

<IDENTIFIER> => <LETTER><IDENTCHARS>

**IDENTIFIER** holds all variables. These variables can only begin with letters.

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

| <DIGIT><IDENTCHARS>

| <UNDERSCORE><IDENTCHARS>

| <HYPHEN><IDENTCHARS>

| ε

**IDENTCHARS** are all possible characters used in identifiers. These include -, \_, alphabets and digits.

LETTER => [A-Za-z]

UNDERSCORE => \_

HYPHEN => -

ε => Empty

<LOGIC\_OP> **=>** <AND> | <OR> | <IMPLIES>

**LOGIC\_OP** holds all connectives except negation.

AND => /\

OR => \/

IMPLIES => ->

<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)

<PREDEFINED\_LOGIC> **=>** <BOOLEAN> | <CONSTANT> | <IDENTIFIER>

**PREDEFINED\_LOGIC** can be a constant, a boolean value or an identifier (a variable).

<COMPLEX\_LOGIC> **=>** (LOGIC\_STMT)

| <NEGATION> <LOGIC\_STMT>

| <LOGIC\_STMT> <LOGIC\_OP> <LOGIC\_STMT>

**COMPLEX\_LOGIC** can simply be another LOGIC\_STMT or a negation followed by a LOGIC\_STMT, since only the negation connective can occur before a LOGIC\_STMT, or, a LOGIC\_OP between two LOGIC\_STMTS.

NEGATION => ~

<LOOP\_STMT> **=>** <LOOP>(<CONDITION>)<STATEMENTS><ENDLOOP>

**LOOP\_STMT** is used to repeat statements dependent on given conditions.

LOOP => loop

ENDLOOP => endloop

<IF\_STMT> **=>** <IF\_ONLY> | <IF\_ELSE>

**IF\_STMT** is either IF\_ONLY or IF\_ELSE.

<IF\_ELSE> **=>** <IF>(<CONDITION>)<THEN><IF\_ELSE><ELSE><IF\_ELSE>

| <STATMENTS>

**IF\_ELSE** is an IF\_STMT which is followed by an else.

IF => if

THEN => then

ELSE => else

<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\_STMT which is 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.

GREATER => >

GREATER\_EQ => >=

LESS => <

LESS\_EQ => <=

EQUALS => ==

NOT\_EQ => ~=

<COMPARATORS> **=>** (<ALGEB\_STMT>) | <ALGEB\_LIST>

**COMPARATORS** can either be ALGEB\_STMTS or ALGEB\_LIST 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 ALGEB\_STMTS.

<ASSIGN\_STMT> **=>** <IDENTIFIER><ASSIGN><STMT\_LIST>

**ASSIGN\_STMT** is used to assign a value to an identifier using STMT\_LIST defined below.

ASSIGN => =

<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.

TAKE => take

<OUTPUT\_STMT> **=>** <GIVE><PREDEFINED\_LOGIC>

**OUTPUT\_STMT** is used to output.

GIVE => give

<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.

WHITE\_SPACE => “ “

FUNCTION => function

<PARAM\_LIST> **=>** <IDENTIFIER><COMMA><PARAM\_LIST>

|<IDENTIFIER>

**PARAM\_LIST** contains all the possible parameters of the function.

COMMA => ,

<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.

<FUNCTION\_CALL> **=>**

**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.

**Reserve Words:** function**,** true, false, give, take, if, then, else, loop, endloop  
Our reserve words are kept short and simple with meaningful names to allow for an easy writable process. This also makes understanding the code by programmers easier, supporting readability of the program. Again in this case, the code becomes reliable as there will be less mistakes.

Note: These tokens are all defined above when used in certain BNF forms.