CS315 – Project #1

Group 15

Project: **YAPL** (Yet Another Programming Language)

Members:

Huseyn Allahyarov (21503572) (Sec. 1)

Ibrahim Mammadov (21603109) (Sec. 1)

Bayram Muradov (21503664) (Sec. 1)

BNF Grammar for YAPL

```
<program> ::= <declaration_g>;<func_def> | <func_def><declaration_g>;
               | <func_def> | <declaration_g>;

<func_def> ::= <func_dec> | <func_dec><func_def> | <void_dec><func_def>
               | <void_dec>

<func_dec> ::= <return_type><func_id>(<declaration_f>){<stmts><return><term>;}

<void_dec> ::= void<func_id>(<declaration_f>){<stmts>}
               | void<func_id>(<declaration_f>){<stmts><return>;}
               | void<func_id>(<declaration_f>){<stmts><return>;<stmts>}
               | void<func_id>(<declaration_f>){<return>;}

<stmts> ::= <stmt> | <stmt><stmts>

<stmt> ::= <func_call>; | <expr>; | <dec>; | <set_expr>; | <if_stmt> | <for>
          | <while> | <comment> | <declaration_g>;

<expr> ::= <var_id> = <term> | <var_id> = <term><arit_op><term>
               | <var_id><increment> | <var_id><decrement>
               | <increment><var_id> | <decrement><var_id>

<set_expr> ::= <set_dec> | <set_dec> = <set_op> | <set_dec> = <set>

<set_dec> ::= <type><set_id>

<term> ::= <var_id> | <number> | <double_number> | <func_call> | <string>
          | <character> | <set>

<if_stmt> ::= <matched> | <unmatched>

<matched> ::= if(<bool_expr>) <matched> else <matched>
               | {<stmts>}

<unmatched> ::= if(<bool_expr>) <matched> {<stmts>}
               | if(<bool_expr>) <matched> else <unmatched>

<logic_expr> ::= <term><cond><term> | <term>

<bool_expr> ::= <logic_expr><bool_op><logic_expr>
               | <logic_expr><bool_op><bool_expr>
               | <logic_expr>

<bool_op> ::= <and> | <or>
```

```
<and> ::= &&

<or> ::= ||

<cond> ::= <less> | <more> | <equal> | <less_equal> | <more_equal> | <not>

<less> ::= <

<more> ::= >

<equal> ::= ==

<less_equal> ::= <=

<more_equal> ::= =>

<not> ::= !=

<for> ::= for(<declaration_g>;<bool_expr>;<expr>){<stmts>}
               | for(<expr>;<bool_expr>;<expr>){<stmts>}

<while> ::= while(<bool_expr>){<stmts>}

<set_id> ::= &<var_id>

<set> ::= {<num_set>} | {<string_set>} | {<double_set>} | {<char_set>}
               | {<bool_set>}

<bool_set> ::= <bool>, <bool_set> | <bool> | {<bool_set>}

<num_set> ::= <number>, <num_set> | <number> | {<num_set>}

<double_set> ::= <double_number>, <double_set> | <double_number> | {double_set>}

<string_set> ::= <string>, <string_set> | <string> | {<string_set>}

<char_set> ::= <character>, <char_set> | <character> | {<char_set>}

<number> ::= <pos> | <neg>

<neg> ::= -<pos>

<pos> ::= <digit><pos> | <digit>

<double_number> ::= <number>.<pos>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<comment_string> ::= <char_all><comment_string> | <char_all>

<string> ::= “<char_all><string>” | <char_all><string> | <char_all> | “<empty>”

<character> ::= ‘<char_all>’

<char_all> ::= <char> | % | ` | ... | ~

<char> ::= <letter> | _ | $
```

```
<letter> ::= a | b | c | ... | z | A | B | … | Z
<var_id> ::= <char><id> | <char>
<id> ::= <number><id> | <number> | <char><id> | <char>
<set_op> ::= <set><op><set_op> | <set><op><set> | <set_id><op><set_id>
<op> ::= <uni> | <intsec> | <subt>
<uni> ::= uni
<intsec> ::= intsec
<subt> ::= subt
<set_rel> ::= <set><rel><set_rel> | <set><rel><set> | <set_id><rel><set_rel>
               | <set_id><rel><set_id>
<rel> ::= <sub> | <super>
<sub> ::= <<
<super> ::= >>
<assignment_op> ::= =
<comment> ::= //<comment_string>\n | /*<comment_string>*/
<type> ::= int | double | string | bool | char
<bool> ::= true | false
<input> ::= iin: | cin: | bin: | sin: | din:
<output> ::= iout: | cout: | bout: | sout: | dout:
<arit_op> ::= <plus> | <minus> | <mult> | <div>
<plus> ::= +
<minus> ::= -
<mult> ::= *
<div> ::= /
<leftb> ::= (
<rightb> ::= )
<lefts> ::= [
<rights> ::= ]
<leftc> ::= {
<rightc> ::= }
```

<increment> ::= ++

<decrement> ::= --

<func_id> ::= <var_id>

<declaration_f> ::= <type><var_id>,<declaration_f> | <type><var_id> | <empty>

<declaration_g> ::= <type><declaration_recursive> | <empty>

```
<declaration_recursive> ::= <var_id>, <declaration_recursive>
               | <var_id> | <var_id> = <expr>, <declaration_recursive> |
               | <var_id> = <expr>
```

<empty> ::= ;

<func_call> ::= <func_id>(<parameter>);

```
<parameter> ::= <empty>
               | <var_id>,<parameter>
               | <number>,<parameter>
               | <string>,<parameter>
               | <character>,<parameter>
               | <double_number>,<parameter>
               | <bool>,<parameter>
               | <var_id>
               | <number>
               | <string>
               | <character>
               | <double_number>
               | <bool>
```

<return> ::= return

Detailed language description for YAPL

PROGRAM – Definition of the source code which consists of global variables and functions.

DECLARATION_G – Used for declaration of one or more variables. Can be used for declaring global or local variables.

FUNC_DEF – Definition of a function which consists of return type, identification, argument list and set of statements which describes the function of this function including what it returns.

VOID_DEC – Declaration of a function with the return type *void*. Description is same as the **FUNC_DEF** except for the return type and returning terminal. May contain *return* to stop the execution of function any time.

STMTS – Set of statements.

STMT – Includes all types of statements a code-block may contain, such as *function call, expression, declaration, conditional statements, loops* or *comments*.

EXPR – Defines the expressions allowed in the language. Includes variable/set assignment with terminals or arithmetic operations with terminals.

SET_EXPR – Used for *set declarations, set operations* or *set relations*.

SET_DEC – Used for the declaration of a set which can be initialized or not.

TERM – Stands for *terminal*. Contains *variables, constants, function calls* or *sets.*

IF_STMT – Used for declaration of conditional *if / else* statements. Statements can be *matched* or *unmatched*.

MATCHED – Defines the conditional *if* statements which are matched with an *else* statement and can also be the functional code-block of either statement.

UNMATCHED – Defines the conditional *if* statements which are not matched with an *else* statement and can also be an *else if* statement.

LOGIC_EXPR – Used for the logical expressions that can be used for *conditional statements* or *loops* and may also contain single *terminal*.

BOOL_EXPR – Can be described as a single *logical expression* or a set of *logical expressions* with the Boolean operations such as *and / or.*

BOOL_OP – Defines the Boolean operations such as *and / or*.

AND – Defines the Boolean operation *and.*

OR – Defines the Boolean operation *or.*

COND – Defines the *conditional operations* such as *less (<), greater (>), equal (==), less or equal (<=), greater or equal (=>)* or *not equal (!=).*

LESS – Defines the conditional operator *less* (<).

MORE - Defines the conditional operator *greater* (>).

EQUAL - Defines the conditional operator *equal* (==).

LESS_EQUAL - Defines the conditional operator *less or equal* (<=).

MORE_EQUAL - Defines the conditional operator *more or equal* (=>).

NOT_EQUAL - Defines the conditional operator *not equal* (!=).

NOT - Defines the Boolean complement operator (!).

FOR – Defines the *for* loop which may contain the iterator which has been declared before or declared in the loop definition.

WHILE – Defines the *while* loop.

SET – Defines the several types of *sets* such as *integer sets, Boolean sets, string sets, char sets* or *double sets.*

NUM_SET – *Set* of *integer* numbers.

BOOL_SET – *Set* of *boolean* values.

STRING_SET – *Set* of *strings*.

CHAR_SET – *Set* of *characters*.

DOUBLE_SET – *Set* of *double* numbers.

SET_ID – Defines the identifier of a *set* which should contain & sign as the first element followed by a single or more <u>*allowed*</u> *characters*.

VAR_ID – Defines the variable identifier which should contain an *allowed character* as the first element followed by other allowed characters or digits.

ID – Defines the string of characters composed of *allowed characters* and/or *digits*.

CHAR_ALL – All characters in the ASCII format.

CHAR – Only *allowed characters* which are letters, dollar sign ($) or underscore (_).

DIGIT – Digits (0-9).

NUMBER – Defines the *integer number* which can be *positive* or *negative.*

NEG – Negative integer number which is followed by the minus (-) character.

POS – Positive integer numbers which are composed of digits.

BOOL – Boolean values which can be either *true* or *false.*

DOUBLE_NUMBER – Defines the floating-point numbers which can be negative or positive.

STRING – Defines the string of characters starting and ending with double quotes (“”).

CHARACTER – Single character value between the single-quotes (‘’).

COMMENT_STRING – String of characters without the quotes in beginning and ending.

SET_OP – Defines the set operations done on the sets.

OP – Set operations such as *union, intersection* or *subtraction*.

UNI - Set operation *union.*

INTSEC – Set operation *intersection.*

SUBT – Set operation *subtraction.*

SET_REL – Set relations done on the sets such as *subset* or *superset*.

REL – Set relations such as *subset* or *superset.*

SUB – Set relation *subset* which is defined as << (subset on the left and superset on the right).

SUPER – Set relation *superset* which is defined as >> (superset on the left and subset on the right).

ASSIGNMENT_OP – Assignment operator (=).

COMMENT – Single-line (//) or multi-line (/* */) comments which are the parts of the source code that are not executed.

TYPE – Variable types such as *int, double, char, string* or *Boolean*.

INPUT – Used for entering values. Tokens *iin, cin, din, bin, sin* are intended to take user input in the forms of *integer, character, floating-point number, Boolean value* or *string of characters* respectively.

OUPUT – Used for printing the values. Tokens *iout, cout, dout, bout, sout* are intended to print values in the forms of *integer, character, floating-point number, Boolean value* or *string of characters* respectively.

ARIT_OP – Arithmetic operations such as *addition, subtraction, multiplication* and *division.*

PLUS – Plus sign (+), used for the addition.

MINUS – Minus sign (-), used for the subtraction.

MULT – Asterisk sign (*), used for the multiplication.

DIV – Slash sign (/), used for the division.

LEFTB – Opening brace sign '('.

RIGHTB – Closing brace sign ')'.

LEFTS – Opening square bracket sign '['.

RIGHTS – Closing square bracket sign ']'.

LEFTC – Opening square bracket sign '{'.

RIGHTC – Closing square bracket sign '}'.

INCREMENT – Double-plus signs (++) used for incrementing the variable by 1.

DECREMENT – Double-minus signs (--) used for decrementing the variable by 1.

EMPTY – No character.

FUNC_ID – Function identifier.

DECLARATION_F – Used for declaring the argument list for a function.

DECLARATION_RECURSIVE – Used for declaring variables recursively.

FUNC_CALL – Function call.

PARAMETER – Parameter values which may be followed by the comma (,) character for sending the parameters to the function

RETURN – Token "return" used for returning the value and stop the execution of the function.

Non-trivial tokens

Allowed characters for the language are $, _ and letters which helps the language to remain reliable and be able to differentiate identifiers from other types of strings. Allowed characters are used mainly for the capital character of the identifiers, digits may follow after the first character.

Usage of & as the capital character for the set identifiers helps the language to differentiate between function and variable identifiers which results in better reliability and readability but poor writability.

Single line comments and multi-line comments are intended for better readability, writability and reliability.

Reserved words are usually abbreviations and have been chosen for better readability and reliability but may result in poor writability.

Some tokens have been used to ease the usage in lexical analyzer.

Many tokens and reserved words have been chosen from the C-type language syntax rules for familiarity and reliability.

Note

Some tokens in Lex file and BNF grammar may vary because of the job distribution in the project. But the BNF grammar and rules have been pursued as much as possible.