Bilkent University

Department of Computer Engineering

CS 315 Project 1

A Programming Language for Sets and its Lexical Analyzer

Name of the Language

.Help

Team Members

Arshia Bakhshayesh - 22001468 - Section 01

Tuğberk Dikmen - 21802480 - Section 01

Table of Contents

1. The Complete BNF Description of .Help

1.1 Program

<program> ::= <start><stmts><end>

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

<stmt> ::= <assignment_stmt> | <set_stmt> | <if_stmt> | <print_stmt> | <loop_stmt>|

<comment_stmt> | <function_stmts>

<assignment_stmt> ::= <variable_identifier> <assign_op>

<variable_identifier><end_symbol> |

<variable_identifier> <assign_op> <type><end_symbol> | <variable_identifier>

<assign_op> <expr><end_symbol>

<expr> ::= <logical_expr> | <arithmetic_expr>

<logical_expr> ::= <set_logical> | <variable_logical> |< logical_expr> <logical_symbol>

<set_logical> | < logical_expr> <logical_symbol> <variable_logical>

<variables> := <variable_identifier> | <function_call>

<set_logical> ::= <set_operand>< bool_logical_symbol> <set_operand>

| <set_operand> <bool_logical_symbol>< list>

| <list> <bool_logical_symbol> <set_operand>

| <list> <bool_logical_symbol> <list>

<variable_logical> ::= <variables> <bool_logical_symbol> <variables>

| <variables> <bool_logical_symbol> <type>

| <type> <bool_logical_symbol> <variables>

| <type> <bool_logical_symbol> <type>

<arithmetic_expr>:::=

<numbers> <arithmetic_symbol> <numbers>

| <numbers> <arithmetic_symbol> <variables>

| <variables> <arithmetic_symbol> <numbers>

| <variables> <arithmetic_symbol> <variables>

| <arithmetic_expr> <arithmetic_symbol> <numbers>

| <arithmetic_expr> <arithmetic_symbol> <variables>

<print_stmt> ::= <PRINT> <LP> <type> <RP><END_SYMBOL>

| <PRINT> <LP> <expr> <RP> <END_SYMBOL>

|< PRINT> <LP> <variables> <RP> <END_SYMBOL>

|< PRINT> <LP> <set_operand> <RP> <END_SYMBOL>

| <PRINT> <LP> <list RP> <END_SYMBOL>

1.2 Variable and set

<numbers> ::= <integer> | <double> | <primitive_functions>
<variable_decleration> ::= <type_identifier><variable_identifier><assign_op><type>

<variable_identifier> ::=

<word><generic_symbol><number>|<word><generic_symbol>|<word><number>|<word

>|

<word><generic_symbol><number><variable_identifier>|<word><generic_symbol><var

iable_identifier> | <word><number><variable_identifier>

<type_identifier> ::= <string_identifier>|<int_identifier>|
<double_identifier>|<bool_identifier>

<string_identifier> ::= string

<int_identifier> ::= int

<double_identifier> ::= double

<bool_identifier> ::= bool

<set> ::= <set_symbol> <variable_identifier>

1.3 Types

<word> ::= <letter> | <word><letter>

<letter> ::= ‘a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'| 'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|

'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'| 'U'|'V'|'W'|'X'|'Y'|'Z'

<number> ::= <digit> | <digit><number>

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

<list>::= <type> <comma> <type>

| <type> <comma><variables >

| <variables> <comma> <type>

| <variables> <comma> <variables>

| <list> <comma> <type>

| <list> <comma> <variables>

<integer> ::= [<arithmetic_symbol>] <number> | <cardinality>

<double> ::= [<arithmetic_symbol>] <number> . <number>

<string> ::= <string_symbol><word><string><string_symbol> |

<string_symbol><number><string><string_symbol> | <string

_symbol><symbol><string><string_symbol> |

<bool> ::= True|False

<type> ::= <numbers>|<string>|<bool>

<numbers> ::= <integer>| <double>| <primitive_functions>

1.4 Set Statements

<function_set> ::= <set> <LP> <RP> | <set> <LP> <parameters> <RP>

<set_operand> ::= <set>| <set_compliment>| <function_set>

<set_stmt> ::= <set_assign> | <set_delete> | <set_add> | <set_remove> |

<set_change> | <input_set_stmt> | <output_set_stmt>

<set_assign> ::= <set_operand > <assign_op> <list> <end_symbol> | <set_operand >

<assign_op> <set_operand > <end_symbol>| <set_operand > <assign_op> <set_ops >

<end_symbol>

<set_delete> ::= <set_operand> <set_delete><LP><RP><end_symbol>

<set_add> ::= <set_operand><plus_symbol><type><end_symbol>

|<set_operand><plus_symbol><variables><end_symbol>

|<set_operand><plus_symbol><list><end_symbol>

|<set_operand><plus_symbol><set_ops><end_symbol>

<set_remove> ::= <set_operand><remove_symbol><type><end_symbol>|

<set_operand><remove_symbol><variables><end_symbol> |

<set_operand><remove_symbol><list><end_symbol>|

<set_operand><remove_symbol><set_ops><end_symbol>

<set_change> ::=

<set_operand><set_change_symbol><integer><change_to_symbol><variables><end_

symbol> |

<set_operand><set_change_symbol><integer><change_to_symbol><type><end_symb

ol>|

<set_operand><set_change_symbol><variables><change_to_symbol><variables><end

_symbol>

|

<set_operand><set_change_symbol><variables><change_to_symbol><type><end_sy

mbol>

<set_ops> ::= <set_operand> <set_op_symbols> <set_operand >

| <set_operand> <set_op_symbols <list>

|< list> <set_op_symbols> <set_operand>

| <list> <set_op_symbols> <list>

| <set_operand> <set_op_symbols> <set_ops>

| <list> <set_op_symbols> <set_ops>

<input_set_stmt> ::= <set_operand> <set_in_symbol> console <end_symbol> |

<set_operand> <set_in_symbol> <string> <end_symbol>

<output_set_stmt> ::= <set_operand ><set_out_symbol> console <end_symbol> |
<set_operand > <set_out_symbol> <string> <end_symbol>

<set_op_symbols> ::= <union_symbol>| <intersection_symbol> | <difference_symbol>

1.5 If

<if_stmt> ::= <normal_if> | <if_else> | <if_elif>

<normal_if> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB>

<if_else> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB> else <LCB <stmts>
<RCB> | if <LP> <logical_expr> <RP> <LCB> <RCB> else <LCB> <stmts> <RCB>

<if_elif> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB> <elif_stmt> else
<LCB> <stmts> <RCB>
| if<LP> <logical_exp>r <RP> <LCB> <RCB> <elif_stmt> else <LCB> <stmts> <RCB>

<elif_stmt> ::= elif <LP> <logical_expr> <RP> <LCB> <RCB>
| elif <LP> <logical_expr> <RP> <LCB> <stmts> <RCB>
| <elif_stmt> elif <LP> <logical_expr> <RP> <LCB> <stmts <RCB>
| <elif_stmt> elif <LP> <logical_expr> <RP ><LCB> <RCB>

1.6 Loops

<loop_stmt> ::= <while> | <for> | <do_while>

<while> ::= while <LP> <logical_expr> <RP> <LCB> <stmts><RCB>|while <LP>
<logical_expr> <RP> <LCB><RCB>

<for> ::= <for_stmt> <LCB> <stmts> <RCB>
|<for_stmt> <LCB> <RCB>

<for_stmt> ::= for <LP> <assign_stmt> <logic_expr> <end_symbol> <assign_stmt>
<RP>

<do_while> ::= do<LCB><stmts><RCB>while<LP><logic_expr><RP> |
do<LCB><RCB>while<LP><logic_expr><RP>

1.7 Function Definition and Function Call

<function_stmts > ::= <function_def> | <function_call>
<function_call_stmt> ::= <function_call> <end_symbol>

<function_call> ::= <variable_identifier><LP><parameters><RP> |
<variable_identifier> <LP><RP>

<function_def> ::= <function_sig <LP <parameters <RP <LCB <stmts <return_stmt
<RCB
| <function_sig> <LP> <RP> <LCB> <stmts> <return_stmt> <RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <return_stmt> <RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <stmts>< RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <RCB>
| <function_sig> <LP> <RP> <LCB ><return_stmt> <RCB>
| <function_sig> <LP> <RP> <LCB> <stmts> <RCB>
| <function_sig> <LP> <RP> <LCB> <RCB>

<function_sig> ::= function <variable_identifier>| <set_symbol> function
<variable_identifier>

<parameters> ::= <type_identifier><variable_identifier>|
<parameters><type_identifier><variable_identifier><comma>|
<set> | <parameters> <comma> <set>

<return_stmt> ::= <return><type><end_symbol> | <return><variables><end_symbol>|
<return><set_operand ><end_symbol>| <return><set_ops ><end_symbol>|
<return><expr ><end_symbol>

1.8 Primitive Functions

<primitive_functions> ::= <set_cardinality>

1.9 Comment
<comment_stmt> ::= <single_line_com> | <multi_line_com>

1.10 Symbols

<union_symbol> ::= -U-

<intersection_symbol> ::= -N-

<difference_symbol> ::= -D-

<string_symbol> ::= ‘“‘

<generic_symbol> ::= _ | -

<empty> ::=

<arithmetic_symbol> ::= <plus_symbol>| <minus_symbol>

<bool_logical_symbol> ::= <LTE> | <GTE> | <GT> | <LT> | <not_equal> |

<equality_symbol>

<logical_symbol> ::= <AND> | <OR>

<plus_symbol> ::= +

<minus_symbol> ::= -

<remove_symbol> ::= ~

<assign_op> ::= =

<set_out_symbol> ::= >>>>

<set_in_symbol> ::= <<<<

<set_logical_symbol> := <== | >== | <not_equal>

<set_change_symbol> := ^

<variable_logical_symbol> ::= <LTE> | <GTE | <GT> | <LT> | <not_equal>

<not_equal> ::= !=

<equality_symbol> ::= ==

<space> ::= ‘ ‘

<set_relation_symbol> ::= <subset_symbol> | <superset_symbol>
<subset_symbol> ::= <==
<superset_symbol> ::= >==

<set_symbol> ::= @

<LP> ::= (

<RP> ::=)

<LT> ::= <

<GT> ::= >

<LTE> ::= <=

<GTE> ::= >=

<LCP> ::= {

<RCP> ::= }

<connector> ::= |

<start> ::= start

<end> ::= stop

<return> ::= return

<comma> ::= ,

<end_symbol> ::= ;

<change_symbol> ::= ^

<change_to_symbol> ::= →

<AND> ::= &&

<OR> ::= ||

2. Explanation of BNF

2.1 Program

<program> ::= <start><stmts><end>

This non-terminal is the whole structure of the language. The program starts with the start token and ends with the and token. Between those tokens there are statements to be executed.

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

Stmts is the collection of statements used in this language. It occasionally can be referred to as a code block and can contain other code blocks too besides normal one line statements.

<stmt> ::= <assignment_stmt> | <set_stmt> | <if_stmt> | <print_stmt> | <loop_stmt>| <comment_stmt> | <function_stmts>

A statement can be composed of several statements such as assignment statement, if statement, set statement, print statement.

<assignment_stmt> ::= <variable_identifier> <assign_op> <variable_identifier><end_symbol> |
<variable_identifier> <assign_op> <type><end_symbol> |
<variable_identifier> <assign_op> <expr><end_symbol>

This non-terminal uses the standard convention of initializing or changing the value of a variable by assigning a value or another variable with = to it. There can also be arithmetic operations to it but the type of LHS and RHS must match.

<expr> ::= <logical_expr> | <arithmetic_expr>

An expression can be composed of two expression such as logical expression and arithmetic expression.

<logical_expr> ::= <set_logical> | <variable_logical> |< logical_expr>

<logical_symbol> <set_logical> | < logical_expr> <logical_symbol>

<variable_logical>

Logical expression produces 1 or 0 and can be made of other logical expressions. Each logical expression must contain a logical symbol. This non-terminal covers all types of logical expressions in the language such as set, variable , and logical literals.

<set_logical> ::= <set_operand>< bool_logical_symbol> <set_operand>

| <set_operand> <bool_logical_symbol>< list>

| <list> <bool_logical_symbol> <set_operand>

| <list> <bool_logical_symbol> <list>

It is used to define sets' logical operations which have their own special logical symbols. A logical set operation can include only and only another logical set operation.

<variable_logical> ::= <variables> <bool_logical_symbol> <variables>

| <variables> <bool_logical_symbol> <type>

| <type> <bool_logical_symbol> <variables>

| <type> <bool_logical_symbol> <type>

Variable logical is composed of variable followed by any variable logical symbol. Used to define for variable logical operations.

<arithmetic_expr>::=

<numbers> <arithmetic_symbol> <numbers>

| <numbers> <arithmetic_symbol> <variables>

| <variables> <arithmetic_symbol> <numbers>

| <variables> <arithmetic_symbol> <variables>

| <arithmetic_expr> <arithmetic_symbol> <numbers>

| <arithmetic_expr> <arithmetic_symbol> <variables>

Arithmetic expression composed of inter or double value followed by arithmetic symbol. Used to define for arithmetic logical operations.

<print_stmt> ::= <PRINT> <LP> <type> <RP><END_SYMBOL>

| <PRINT> <LP> <expr> <RP> <END_SYMBOL>

|< PRINT> <LP> <variables> <RP> <END_SYMBOL>

|< PRINT> <LP> <set_operand> <RP> <END_SYMBOL>

| <PRINT> <LP> <list RP> <END_SYMBOL>

Print stmt is a non-terminal that is used for out printing the statements.

2.2 Variable and set

<variable_decleration> ::=
<type_identifier><variable_identifier><assign_op><type>

Variable declaration is a non-terminal that is used for assigning value to variables that pre pre-identified.

<variable_identifier> ::=

<word><generic_symbol><number>|<word><generic_symbol>|<word><nu

mber>|<word>|

<word><generic_symbol><number><variable_identifier>|<word><generic_s

ymbol><variable_identifier> | <word><number><variable_identifier>

A variable identifier is a non-terminal that is used for naming the variables. It is composed of a word or word combined with symbols or numbers.

<type_identifier> ::= <string_identifier>|<int_identifier>|
<double_identifier>|<bool_identifier>

Type identifier non-terminal is defined. It might be a string_identifier, int_identifier, double_identifier, bool_identifier.

<string_identifier> ::= string

This non-terminal is defined for recognizing the “string” expression.

<int_identifier> ::= int

This non-terminal is defined for recognizing the “int” expression.

<double_identifier> ::= double

This non-terminal is defined for recognizing the “double” expression.

<bool_identifier> ::= bool

This non-terminal is defined for recognizing the “bool” expression.

<set> ::= <set_symbol> <variable_identifier>

This non-terminal is composed of two things, set symbol (@) and a variable identifier, this non-terminal used widely since in our language, sets are mandatory to be used with set symbol whenever they are called.

2.3 Types

<word> ::= <letter> | <word><letter>

<letter> ::= ‘a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|

'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|

'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|

'U'|'V'|'W'|'X'|'Y'|'Z'

Letters are defined in this non-terminal.

<number> ::= <digit> | <digit><number>

The non-terminal number is composed of digits.

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

Digits are defined in this non-terminal.

<list>::= <type> <comma> <type>

| <type> <comma><variables >

| <variables> <comma> <type>

| <variables> <comma> <variables>

| <list> <comma> <type>

| <list> <comma> <variables>

The list non-terminal is used widely to initialize sets. Lists can be one of the following types: number list, string list or double list.

<integer> ::= <arithmetic_symbol> <number> | <cardinality>

Integer non-terminal is basically a number or a cardinality.

<double> ::= <arithmetic_symbol> <number> . <number>

Double non-terminal is the combination of two numbers with a dot.

<string> ::= <string_symbol><word><string><string_symbol> |

<string_symbol><number><string><string_symbol> | <string

_symbol><symbol><string><string_symbol> |

Strings can be composed of words, symbols, numbers or any combination of these with string symbol (“) at the beginning and at the end.

<bool> ::= True|False

Bool non-terminal is defined. It might be True or False.

<type> ::= <numbers>|<string>|<bool>

<numbers> ::= <integer>| <double>| <primitive_functions>

Type non-terminal is defined. It might be an integer, double, string or bool.

2.4 Set Statements

<set_stmt> ::= <set_assign> | <set_delete> | <set_add> | <set_remove> |

<set_change> | <input_set_stmt> | <output_set_stmt>

A set statement can be an initialization, assignment, deletion or a set operation.

<set_operand> ::= <set>| <set_compliment>| <function_set>

The initialization of a set is composed of <set> non-terminal and an end symbol.

<set_assign> ::= <set_operand > <assign_op> <list> <end_symbol> |

<set_operand > <assign_op> <set_operand > <end_symbol>| <set_operand

> <assign_op> <set_ops > <end_symbol>

To assign a set, a list must be provided with curly brackets.

<function_set> ::= <set> <LP> <RP> | <set> <LP> <parameters> <RP>

Set functions are defined by this non-terminal.

<set_delete> ::= <set_operand> <set_delete><LP><RP><end_symbol>

This non-terminal is used to define built-in set delete functions.

<set_add> ::= <set_operand><plus_symbol><type><end_symbol>

|<set_operand><plus_symbol><variables><end_symbol>

|<set_operand><plus_symbol><list><end_symbol>

|<set_operand><plus_symbol><set_ops><end_symbol>

This non-terminal is used to define to add new variable to an existing set.

<set_remove> ::= <set_operand><remove_symbol><type><end_symbol>|

<set_operand><remove_symbol><variables><end_symbol> |

<set_operand><remove_symbol><list><end_symbol>|

<set_operand><remove_symbol><set_ops><end_symbol>

This non-terminal is used to define to remove a variable from an existing set.

<set_change> ::=

<set_operand><set_change_symbol><integer><change_to_symbol><varia

bles><end_symbol> |

<set_operand><set_change_symbol><integer><change_to_symbol><type>

<end_symbol>|

<set_operand><set_change_symbol><variables><change_to_symbol><vari

ables><end_symbol>

|

<set_operand><set_change_symbol><variables><change_to_symbol><typ

e><end_symbol>

Our language utilizes this non terminal to change a specified element in the set.

<set_ops> ::= <set_operand> <set_op_symbols> <set_operand >

| <set_operand> <set_op_symbols <list>

|< list> <set_op_symbols> <set_operand>

| <list> <set_op_symbols> <list>

| <set_operand> <set_op_symbols> <set_ops>

| <list> <set_op_symbols> <set_ops>

Some set operations are introduced in this non-terminal. There are set functions for union, intersection, difference and complement.

<input_set_stmt> ::= <set_operand> <set_in_symbol> console

<end_symbol> | <set_operand> <set_in_symbol> <string> <end_symbol>

This non-terminal is defined for taking input to a set from a file or command prompt.

<output_set_stmt> ::= <set_operand ><set_out_symbol> console
<end_symbol> | <set_operand > <set_out_symbol> <string> <end_symbol>

This non-terminal is defined for outputting a set into a file or command prompt.

2.5 If

<if_stmt> ::= <normal_if> | <if_else> | <if_elif>

Combination of the if statements.

<normal_if> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB>

An if statement defined. Forcing the usage of brackets solves the dangling else problem.

<if_else> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB> else <LCB> <stmts> <RCB> | if <LP> <logical_expr> <RP> <LCB> <RCB> else <LCB> <stmts> <RCB>

Else statement defined in order to implement other conditions rather than what wanted in if.

<if_elif> ::= if <LP> <logical_expr> <RP> <LCB> <stmts> <RCB> <elif_stmt> else <LCB> <stmts> <RCB>
| if<LP> <logical_exp>r <RP> <LCB> <RCB> <elif_stmt> else <LCB> <stmts> <RCB>

Else if statement is defined in order to represent specific else cases.

<elif_stmt> ::= elif <LP> <logical_expr> <RP> <LCB> <RCB>
| elif <LP> <logical_expr> <RP> <LCB> <stmts> <RCB>
| <elif_stmt> elif <LP> <logical_expr> <RP> <LCB> <stmts <RCB>
| <elif_stmt> elif <LP> <logical_expr> <RP ><LCB> <RCB>

Helper for else if is defined.

2.6 Loops

<loop_stmt> ::= <while> | <for> | <do_while>

Loop non-terminal is defined. It might be a while, for or do-while loop.

<while> ::= while <LP> <logical_expr> <RP> <LCB> <stmts><RCB>|while <LP> <logical_expr> <RP> <LCB><RCB>

While non-terminal is defined with the while keyword and logical expression with parentheses. A while non-terminal might contain statements.

<for> ::= <for_stmt> <LCB> <stmts> <RCB>
|<for_stmt> <LCB> <RCB>

For non-terminal is defined with a for statement and for non-terminal might contain statements.

<for_stmt> ::= for <LP> <assign_stmt> <logic_expr> <end_symbol> <assign_stmt> <RP>

For statement is composed of a keyword and an assignment statement, logical expression, and one more assignment statement in parentheses.

<do_while> ::= do<LCB><stmts><RCB>while<LP><logic_expr><RP> | do<LCB><RCB>while<LP><logic_expr><RP>

This non-terminal is composed of do and while keywords. It can contain statements and a logical expression is provided for looping.

2.7 Function Definition and Function Call

<function_stmts > ::= <function_def> | <function_call>

Function non-terminal is composed of either function definition or function call.

<function_call> ::= <variable_identifier><LP><parameters><RP> | <variable_identifier> <LP><RP>

To call a predefined function, a variable name should be used to call the function.

<function_def> ::= <function_sig <LP <parameters <RP <LCB <stmts <return_stmt <RCB
| <function_sig> <LP> <RP> <LCB> <stmts> <return_stmt> <RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <return_stmt> <RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <stmts>< RCB>
| <function_sig> <LP> <parameters> <RP> <LCB> <RCB>
| <function_sig> <LP> <RP> <LCB ><return_stmt> <RCB>
| <function_sig> <LP> <RP> <LCB> <stmts> <RCB>
| <function_sig> <LP> <RP> <LCB> <RCB>

To define a function, ‘function’ keyword must be used with a type and a variable identifier for naming.

<parameters> ::= <type_identifier><variable_identifier>|

<parameters><type_identifier><variable_identifier><comma>|
<set> | <parameters> <comma> <set>

Parameters are defined which consist of variable type and variable identifier itself.

<return_stmt> ::= <return><type><end_symbol> |
<return><variables><end_symbol>|
<return><set_operand ><end_symbol>| <return><set_ops ><end_symbol>|
<return><expr ><end_symbol>

Return statement is defined in order to return variables (the product of function).

2.8 Primitive Functions

<primitive_functions> ::= <set_cardinality>

Primitive functions are composed of cardinality.

2.9 Comment

<comment_stmt> ::= <single_line_com> | <multi_line_com>

Comment selector composed of single line comment and multi line comment.

2.10 Symbols

<union_symbol> ::= -U-

Definition of union symbol is provided.

<intersection_symbol> ::= -N-

Definition of intersection symbol is provided.

<difference_symbol> ::= -D-

Definition of difference symbol is provided.

<set_op_symbols> ::= <union_symbol>| <intersection_symbol> | <difference_symbol>

Combination of the set operation symbols.

<comlement_symbol> ::= ~

Definition of complement symbol is provided.

<string_symbol> ::= ““

Definition of string symbol is provided.

<generic_symbol> ::= _ | -

Some generic symbols are defined to provide implementation for other non-terminals such as <identifier>.

<empty> ::=

Empty non-terminal is used to help the definition of other non-terminals.

<plus_symbol> ::= +

Symbol for plus operation is defined.

<remove_symbol> ::= ~

Symbol for remove operation is defined.

<assign_op> ::= =

Symbol for assignment statements is defined.

<set_out_symbol> ::= >>>>

Symbol for outputting sets is provided.

<set_in_symbol> ::= <<<<

Symbol for inputting sets is provided.

<set_logical_symbol> := <== | >== | <not_equal>

Combination of all set logical symbols are defined.

<bool_logical_symbol> ::= <LTE> | <GTE> | <GT> | <LT> | <not_equal> |

<equality_symbol

Combination of all variable logical symbols are defined.

<not_equal> ::= !=

Not equal symbol is defined.

<equality_symbol> := ==

Equality symbol is defined.

<logical_symbol> ::= <AND> | <OR>

Logical “and” and “or” symbols are defined.

<AND> ::= &&

And symbol is defined.

<OR> := ||

Or symbol is defined.

<set_relation_symbol> ::= <subset_symbol> | <superset_symbol>
<subset_symbol> ::= <==
<superset_symbol> ::= >==

Set relation symbols are defined.

<set_symbol> ::= @

Set symbol is defined.

<LP> ::= (

Symbol for left parenthesis is defined.

<RP> ::=)

Symbol for right parenthesis is defined.

<LT> ::= <

Symbol for less than is defined.

<GT> ::= >

Symbol for greater than is defined.

<LTE> ::= <=

Symbol for less than equal is defined.

<GTE> ::= >=

Symbol for greater than equal is defined.

<LCP> ::= {

Symbol for left curly brackets is defined.

<RCP> ::= }

Symbol for right curly brackets is defined.

<connector> ::= |

Symbol for the vertical line is defined.

<start> ::= start

Symbol to start the program is defined.

<end> ::= stop

Symbol to stop the program is defined.

<return> ::= return

Symbol for the “return” is defined.

<comma> ::= ,

Symbol for the comma is defined.

<end_symbol> ::= ;

Symbol for the end is defined.

<change_symbol> ::= ^

Symbol for the change is defined.

<change_to_symbol> ::= -->

Symbol for the change to is defined.

<LBR> ::=

Symbol for the left bracket is defined.

<RBR> ::=

Symbol for the right bracket is defined.

3. Nontrivial Tokens

3.1 Literals

In our language .Help there are five different literals to use for different purposes. Followed as; “int”, “double”, “char”, “string” and “bool”.

int: int values are signed integer numbers that are not followed by “.” character.

double: double values are signed numbers that have “.” character after itself and followed by unsigned integer value. Although this implementation of “int” and “double” hardens the writability, it increases the readability and reliability of our language as this implementation erases the confusion of number types.

string: string literals are defined as any character or characters on ASCII table with quotation marks both at the start and at the end of character. .Help language also accepts a single character as a string. The reason behind that is to increase the writability of the program, so that users don’t have to worry about what symbol to use in a single character or group of characters.

bool: bool is used for boolean which can be defined as True or False.

3.2 Identifiers

In .Help language identifiers cannot start with a number or any non alphabetic character. Identifiers should start with an alphabetic character, after that identifier can be followed by any number. Furthermore, in order to increase the readability and writability, users can optionally use “-” or ”_” between the alphabetic and the number part of the identifier. Our main goal is to offer the user the most comfortable of defining identifiers while increasing both readability and writability.

3.3 Reserved Words

In .Help language reserved words are similar to other popular languages reserved words like C++ and Java. The common reserved words on .Help are “if”, “else”, “for”, “while” and “do”. We wanted them to be similar to other languages in order to decrease the adjustment time of our users to .Help language. Furthermore, this implementation increases the readability and writability, as the new users know what those words stand for. In addition to that, .Help has a different style of function operations. We added “function” as a reserved word at the beginning of function definitions or function calls, by doing

this the readability of .Help raised up. The functions are now easily noticeable along the other parts of the code.

3.4 Comments

In .Help language comments are designed to increase the writability of the user. .Help offers users to both implement a single line of comment and multi line of comments. In single line comments the only thing that user should do is add a “#” sign before the comment line, when the new line comes the comment ends as well. For multi line comments the implementation is very similar, the first line of comment start with “##” and end with “##”. Multi line comments offer users to write as long as they want without restricting them to only one line.

4. Evaluation of .Help

4.1 Readability

.Help is a fundamental language that focuses on sets and set operations while offering the other futures like loops, conditions and functions.

In order to maintain and increase readability of .Help identifiers are supported with the “-” or “_” characters. Those characters are only used in identifiers, so that while a user trying to find an identifier “-” , “_” characters made them easy to spot. Furthermore, for both literal names and reserved words names are selected to refer to the background knowledge of the user. With that user could easily understand which literal name or reserved word has the same future in well known languages like C++, Java. Moreover, comment lines and the block of comments helps most to understand the program. By thinking that we added both single line and multiple line comments to .Help language and made them easy to read with using “#” character for single line comment and “##” for multiple lines of comments. In addition to that, sets use the “@” sign at the beginning of it. As the .Help language does not use any specific character beginning of identifiers or reserved verbs, the sets outshine the others and are easy to notice. The simplicity that .Help offers, increases the readability of the language.

4.2 Writability

All of the non trivial tokens that .Help language contains have a lot of contribution to increase the writability of the language. In literals .Help uses string for both char and string values so that users do not need to worry about. With that future the conversation issues will not occur. Furthermore, the rule for identifiers that .Help language has gives freedom to users. Users can define a new variable without worrying about which symbol he/she should use and finish the variable name however they want. In addition to that, .Help language has common reserved words that are used in other languages, with this users could easily code their program without thinking of another unnecessary reserved word. Finally, as .Help language is a simple language that mostly focuses on sets and set operations, we want to make sets both outstanding and easy to write at the same time. So to define a set the only thing that the user should do is put a “@” sign before it. Not only does it increase the writability, it also contributes to the readability of the program. With all of those futures .Help has a great context which makes it easy to write.

4.3 Reliability

Like all other languages, reliability is the most important thing that a language should have in order to perform the same under all circumstances. In order to create this reliability the readability and writability of the language plays the greatest roles. As those defined above .Help language is a simple language that offers strong writability and readability. In addition to what is mentioned, .Help does not give access to memory locations of variables to users and does not have pointer based structure that can complicate things for use. Furthermore it is not possible for users to refer to the same memory address using the different variable name. Combination of all of those makes .Help a reliable language.