Bilkent University CS315 Project Report

Superset

Spring 2019-2020

Full Name	Student ID	Section
Batuhan Özçömlekçi	21703297	Section 01
Bora Kurucu	21703404	Section 01
Fatih Karahan	21704159	Section 01

Bilkent University, April 8, 2020

Contents

1 BNF of The Language

```
<program> -> <list>

<list>  → <declaration> | <list> <declaration>
<list> →  <list_basic> | <list> <list_basic>
<list_basic> → <func_decl> | <stmt>

<func_decl> → FUNCTION IDENTIFIER LP <argument_list RP> <block>
            | FUNCTION IDENTIFIER LP RP <block>

<argument_list> → <primary> | <argument_list> COMMA <primary>

<stmt> →  <expr_stmt>  | <loop_stmt> | <return_stmt> | <block> | <set_pull> | <set_push>
          | <print_stmt> | <delete_stmt> | <assn_stmt> |  <if_stmt>

<block> → LB <list> RB

<delete_stmt> → SET_IDE ARROW SET_DELETE LP RP SC

<return_stmt> →  RETURN expr_stmt

<if_stmt> → IF LP <expr> RP <stmt> %prec LESS_ELSE
            | IF LP <expr> RP <stmt> ELSE <stmt>

<loop_stmt> → <for_stmt> | <while_stmt>

<while_stmt> → WHILE LP   <expr>   RP <stmt>

<for_stmt> → FOR LP <assn_stmt>  <expr_stmt>  <expr> RP <stmt>
            | FOR LP <assn_stmt>  SC  <expr> RP  <stmt>
            | FOR LP <assn_stmt>  <expr_stmt>  <assn> RP <stmt>
            | iterative_for

<iterative_for> →  FOR LP IDENTIFIER COLON SET RP <expr_stmt>
                | FOR LP IDENTIFIER COLON SET_IDE RP block
                |  FOR LP IDENTIFIER COLON SET RP block

<print_stmt> → PRINT LP <expr> RP SC | PRINT LP STRING RP SC | PRINT LP STRING_IDE RP SC

<contain_expr> → SET_IDE ARROW SET_CONTAINS LP <argument_list> RP

<assn_stmt> → <assn> SC
<assn> → <string_assn> | <set_assn> | <number_assn>

<expr_stmt> → <expr> SC

<expr> → <number_expr> | <set_expr>

<string_assn> → STR_IDE ASSN_OP STRING
            | STR_IDE ASSN_OP INPUT LP RP
            | STR_IDE ASSN_OP INPUT LP STRING RP
            | STR_IDE ASSN_OP STR_IDE
            | FUNC STR_IDE LP <argument_list> RP
```

```
            | FUNC STR_IDE LP RP

<number_assn> → IDENTIFIER ASSN_OP <number_expr>
<number_expr> →  <number_expr> <general_comp_op> <number_addt> | <number_addt>
<number_addt> →  <number_addt> <basic_addition_op> <number_mult> | <number_mult>
<number_mult>  →  <number_mult> <basic_multiplication_op> <number_call> | <number_call>
<number_call> →  FUNC IDENTIFIER LP <argument_list> RP | FUNC IDENTIFIER LP RP | <number_basic>
<number_basic>  →  NUMBER | IDENTIFIER | LP <number_expr> RP
                | <set_size> | INPUT LP RP | INPUT LP STRING RP
                | <contain_expr>


<set_assn> → SET_IDE ASSN_OP <set_expr> | IDENTIFIER ASSN_OP <set_expr>
<set_expr> →  <set_expr>  <general_comp_op> <set_arith> | <set_arith>
<set_arith> →  <set_arith> <set_arith_op> <set_unary> | <set_unary>
<set_unary> →   <set_unary_op> <set_basic> | <set_call>
<set_call> → FUNC SET_IDE LP <argument_list> RP | FUNC SET_IDE LP RP | <set_basic>
<set_basic> →  <set> | SET_IDE | LP <set_expr> RP


<set> → SET_INIT | LB <argument_list> RB

<set_pull>  → SET_IDE PULL_OP <primary> SC
<set_push> → SET_IDE PUSH_OP <primary> SC
<set_size>  → CARDINALITY_OP SET_IDE  | CARDINALITY_OP SET

<general_comp_op> → LT | LEQ | GT | GEQ | EE | NE | OR | AND

<set_arith_op> → UNION_OP | INTERSECTION_OP |  DIFF_OP  | CARTESIAN_OP
<set_unary_op> → POWERSET_OP

<basic_addition_op> → PLUS_OP|MINUS_OP
<basic_multiplication_op> → MULTIPLY_OP|DIVIDE_OP|MOD_OP

<primary> → NUMBER | STRING | IDENTIFIER | SET_IDE |  STR_IDE | SET
```

2 Description of Non-Terminal Literals

$< program >$

The program is the starting literal of the language. It includes a list. This literal is named according to the conventions.

$< list >$

The list is either a single list_basic or a declaration followed by a list_basic. It has a left recursive property.

$< list\_basic >$

list_basic is either a single function declaration or a statement.

$< func\_decl >$

The declaration of functions starts with a FUNCTION token followed an identifier, name, of the function and then a list of arguments that are encapsulated by parentheses. After this signature of the function, there is a block of statements where the inputted arguments is going to be used.

$< argument\_list >$

Argument list is a list of identifiers separated by a comma. It has a left recursive property.

$< stmt >$

Statements are the literals of some main type of operations in this language. They are either expressions that are written in order to handle fundamental mathematical operations, set operations; printing operation; logical operations; loops; the operation of returning from a function or a block of statement that might include all of these previously discussed. The statements are the parts of the language that doesn't have a value in itself, while having side effects.

$< block >$

Block is a statement encapsulated by curly brackets.

$< delete\_stmt >$

Delete stmt is used to delete the set from the program

$< return\_stmt >$

Return statement is used to save and return a value at the end of the performed functions. It returns an expression statement which might a value or a set of values. RETURN token should be inserted to the start of the expression that is wanted to be returned.

$< if\_stmt >$

If statement is the basic conditional statement to control the flow of the program. It performs different actions depending on the expression given to it. It can also have an optional else statement afterwards. The way the dangling else problem was solved is by giving the if statement without else a smaller precedence than an if statement with an else.

loop_stmt >

Loops are either for loops or while loops in this language.

< *while_stmt* >

While loops are the classic loops including a logic expression after WHILE. The while loop loops the block or the expression coming after them until the logical expression turns out to be false.

< *for_stmt* >

For loops are the enhanced version of while loops where you can use assignment statements (initializations) and expressions inside the parenthesis. A for statement is either normal for loops where it loops the block or the expression after it or the iterative (enhanced) for loop.

< *iterative_for* >

Iterative for loop does the same job with normal for loops yet its definition is made so that for every element (corresponds to first IDENTIFIER) in the group, the iterative for loop processes the expression statement or the block after for loop. The groups including the elements can be a SET or an IDENTIFIER.

< *print_stmt* >

Basically, the print statement prints the expression encapsulated by parentheses. There is a semicolon at the end of the statement.

< *contain_expr* >

Containment expression checks whether some argument is included in the set or not

< *assn_stmt* >

Assignment statement is a assignment followed by a semicolon.

< *assn* >

Assignment without a semicolon, it is required for for loops.

< *expr_stmt* >

Expression statement is an expression followed by a semicolon.

< *expr* >

Expression is either a number expression (operations including numbers) or set expression (operations including sets)

< *string_assn* >

The string assignment includes an identifier consisting of alphanumeric characters which is assigned to a STRING primitive type by the use of assignment operator. This primitive type can be called by an identifier, by input, by function return or directly

< *number_assn* >

The number assignment includes an identifier consisting of alphanumeric characters which is assigned to a numeric expression's result in the right hand side (it can be a series of operations) by the use of assignment operator.

$< number\_expr >$

The number expression is either a single number addition or a number expression followed by a number addition and a general comparison operator between them to make comparisons between these two expressions. It is written left recursive and the definition ensures addition of numbers have the precedence over number expressions (i.e. comparison operators)

$< number\_addt >$

The number addition is either a single number multiplication or number addition followed by a number multiplication and a number addition operator between them. By this left recursive implementation, the number expressions including addition is separated from the number expressions including multiplications so that it ensures that multiplications are done first. Likewise, it indicates that multiplication operators (like multiplication, division) have operator precedence over addition operators (like addition, subtraction operator).

$< number\_mult >$

The number multiplication is either a single number factor or number multiplication followed by a number factor and a number multiplication operator between them. By this left recursive implementation, the number expressions including multiplication is separated from the number expressions including factorial so that it ensures that multiplications are done first. Likewise, it indicates that expressions encapsulated by parentheses have operator precedence over multiplication operators (i.e. multiplication and division operator).

$< number\_call >$

Used to call a function, to make assignments such as x = f1(a,2,3); possible

$< number\_factor >$

Basic values that an assignment can get, e.g x = 3; x = y; x = (another expression); x = #set

$< set\_assn >$

The set assignment includes an identifier consisting of alphanumeric characters which is assigned to the result of a set expression by the use of assignment operator.

$< set\_expr >$

The set expression is either a single set arithmetic or a set expression followed by a set union and a general comparison operator between them to make comparisons between these two expressions. It is written left recursive and the definition ensures union of sets have the precedence over set expressions (i.e. comparison operators)

$< set\_arith >$

The set arithmetic expression is either a single unary set expression or a set arithmetic expression followed by a set unary expression, separated with a set arithmetic operator(union, difference etc.). By this left recursive implementation, the set expressions including arithmetics is separated from the set unary expressions so that it ensures that unary expressions are done first. Likewise, it indicates that unary operators (like powerset,

cardinality) have operator precedence over arithmetic operators (like union, intersection, difference operator).

$< set\_unary >$

The set unary expression is either a set unary operator followed by a set(identifier or constant), or a set function call. This ensures that function calls have the highest precedence amongst set operations.

$< set\_call >$

The set call expression is either a function call with the format function_name(argument_list), or a basic set.

$< set\_basic >$

The set basic is either a constant set, a variable identified with the identifier, a set expression inside parenthesis or an input set

set Definition of set is made in here, a set is arguments encapsuled by curly brackets

$< set\_pull >$

Set pull removes an from a set. e.g set1 >> 5, removes 5 from the set1.

$< set\_push >$

Set push pushes a new element to a set. e.g set1 << 5, adds 5 to the set1.

$< set\_size >$

Set size is got by using the cardinality operator. Cardinality operator followed by a set or an identifier returns the number of elements of a set.

$< general\_comp\_op >$

General comparison operators to compare numbers and sets. For sets, subset and superset logic is used.
e.g. if set1 < set2, set1 is subset of set2
e.g. if set1 >= set2, set1 is superset or equal to set2

$< set\_arith\_op >$

Arith op is the set of arithmetic ops for sets. Two sets are may be unified,intersected, or subtracted depending on the operation.

$< set\_unary\_op >$

Unary operations include power set operation and power set operation creates a set including all subsets of the set that the operator is applied.

$< basic\_addition\_op >$

There are mathematical addition operation, subtraction operation under the (identifier) section of addition operation.

$< basic\_multiplication\_op >$

There are mathematical multiplication operation, division operation, modulo operation under the (identifier) section of multiplication operation.

$< primary >$

Primary values can be a NUMBER, a STRING, a SET or an IDENTIFIER

3 Terminals

< smaller than operator ,also used in comparing two sets as a subset relationship e.g. if set1 < set2, set1 is subset of set2

> greater than operator,also used in comparing two sets as a superset relationship e.g. if set1 > set2, set1 is superset of set2

<= smaller than or equal to operator,also used in comparing two sets as a subset or equal relationship e.g. if set1 <= set2, set1 is subset of set2

>= greater than or equal to operator

== equal operator

! = not equal to operator

= assignment operator

: + union operator for sets

: − difference operator for sets

: & intersection operator for sets

<< operator to add an element to a set

>> operator to remove an element from a set

operator to get number of elements in a set

∧ operator to get the powerset of a set

% common mod operator

+ operator used in numerical addition

− operator used in numerical subtraction

∗ operator used in numerical multiplication

/ operator used in numerical division

, used to separate argument lists of functions

() parentheses used to group expressions

{} curly brackets used to declare sets

&& AND operator in bool expressions

|| OR operator in bool expressions

// comment operator

? signature to indicate function calls

− > signature to call set built-in functions calls

$ operator used to denote set variables

∼ operator used to denote string variables

4 Nontrivial Tokens

4.1 Comments

Comments in this language starts with "//" and span a single line. Every word in the line beginning with "//" is interpreted as a comment. Our motivation to put comments feature to this language was to ensure the simplicity of the language. This simplicity increased both the readability, writability and reliability of the language. Especially, comments are useful components to increase readability and writability of a programming language by highlighting the important parts or discounting the unnecessary issues.

4.2 Identifiers

Identifiers start with a letter followed by a number of alphanumeric characters as in C based languages. Our motivation was to increase the well-defined quality of the identifiers. This well defined quality of identifiers helps to increase readibility and writability of the language significantly. $\mathrm{SET}_IDEisusedforsetidentifiers, STR_IDEisusedfo$

identifier : thistoo123123, ?thismaybe2213as123well(args)

identifier : $thistoo123123, ?$thismaybe2213as123well(args)

identifier : $\sim thistoo123123, ? \sim thismaybe2213as123well(args)$

4.3 Literals

number: Number can be real or an integer, this way simplifies to conversion issues between doubles and integers in most of the languages. Although this property helps with the simplicity as well as the writability and the readability of the language, it might create issues about data types.

string: A stream of characters inside two quotation marks. Quotation marks used to avoid confusion with identifiers. The motivation to use string data types in this language was to increase functionality of the language by increasing abstraction and simplicity. This property of strings increases the simplicity by introducing a new data type to the language.

set: Either true, or false. This property of booleans increases the simplicity by introducing a new data type to the language. The set expressions adds more functionality and readability to the language.

4.4 Reserved Words

while	Used in while loop. It introduces a highly beneficial functionality.
for	Used in for loop. It introduces a highly beneficial functionality.
return	Used in return operations in functions. It introduces a highly beneficial functionality.
print	Used to print some text or value to console. It increases simplicity by abstraction.
if	Used in if-if/else statements. It helps to create logical control flows.
else	Used in if/else statements. It helps to create logical control flows.
function	Used to declare functions. It helps to highlight function definitions.
setContains	Word for checking whether an element exist in a set
setDelete	Word for deleting a set
input	Word for getting an input

5 Evaluation of Language Design Criterias

5.1 Readability

The name of the literals and their positions are chosen such that the design conventions are similar to most of the commonly used programming languages or mathematical expressions defining an operation. This property of the language increases the readability of the language. Moreover, the language tries to demonstrate as little unnecessary detail as it can to the user of this language so that it achieves to be simple. Hence the language enhances its readability by the help of simplicity yet it is a trade off between reliability and readability. For example, variable declarations does not specify which primitive data type the identifier is going to be assigned.

5.2 Writability

Writability is highly increased with the merge of integers and doubles to a single type: number. In that way common conversation issues will not occur, hence the user won't need to worry. Also common loop structures for and while are added (i.e. enhanced for loop). Furthermore, this language does not require type declarations for strings,numbers and sets, like in python. This enables users to code without worrying about the matching,etc. On top of that, set operators are very similar to the normal arithmetic operators, only denoted as set operators b having a colon(:) before them. The operators are tried to be used as similar to their original mathematical definitions and thus, writability of expressions is increased in this way.

5.3 Reliability

Recursive definition for some of the non-terminal literals are constructed such that it is either right recursive or left recursive which prevents the possible ambiguity in the language as well as multiple options for the choice of a parse tree of a program written in this language. However, avoiding ambiguity wasn't the main purpose while writing this programming language. Therefore, there could be ambiguous parts for the sake of simplicity. Moreover, reliability of the language could be affected by the absence of data type specifications in the variable declarations. Besides, issues like type checking, exception handling haven't been considered and it might have an impact on the reliability issue.