CS-315 Project 1 Report

Duygu Nur Yaldız 21702333 Section-2

Arda Göktoğan 21702666 Section-2

The name of our language is lang.

The Complete BNF Description of lang:

```
<program>             → <function_list>
<function_list>       → <function>
                      | <comment>
                      | <function> <function_list>
                      | <comment> <function_list>
<function>            → <type> <identifier> ( <param_list> ) { <stmt_list> }
<param_list>          →
                      | <type> <identifier> , <param_list>
<stmt_list>           → <stmt>
                      | <comment>
                      | <comment> <stmt_list>
                      | <stmt> <stmt_list>
<stmt>                → <assign_stmt>
                      | <decl_stmt>
                      | <del_stmt>
                      | <if_stmt>
                      | <loop_stmt>
                      | <return_stmt>
                      | <function_call_stmt>
                      | <write_stmt>
                      | <read_stmt>
<assign_stmt>         → <identifier> = <expr> ;
```

```
<expr>              → <expr> + <term>
                    | <expr> - <term>
                    | ( <expr> )
                    | <term>
                    | <function_call>
<term>              → <set> & <term>
                    | <identifier> & <term>
                    | <identifier>
                    | <set>
<set>               → { <element_list> }
<element_list>      →
                    | <element> , <element_list>
<element>           → <number>
                    | <string>
                    | <set>
                    | <bool>
                    | <identifier>
<decl_stmt>         → <type> <identifier_list> ;
<identifier_list>   → <identifier>
                    | <identifier> , <identifier_list>
<del_stmt>          → delete <identifier> ;
<if_stmt>           → if ( <logic_expr> ) <stmt_list> end
                    | if ( <logic_expr> ) <stmt_list> else <stmt_list> end
<logic_expr>        → <logic_expr> || <logic>
                    | <logic_expr> && <logic>
                    | <logic>
                    | ( <logic_expr> )
<logic>             → <expr> <relation_op> <expr>
                    | <bool>
```

```
<relation_op>            → <
                         | >
                         | ==
                         | ?
                         | !=
<loop_stmt>              → while ( <logic_expr> ) { <stmt_list> }
<return_stmt>            → return <element> ;
<function_call_stmt>     →<function_call> ;
<function_call>          → <identifier> ( <input_list> )
<input_list>             → <element> <input_list>
                         |
<write_stmt>             → write <element>;
<read_stmt>              → read <identifier>;
<type>                   → set
                         | number
                         | string
                         | bool
<identifier>             → <alphabetic>
                         | <alphabetic><alphanumeric>
<number>                 → <digit>
                         | <digit><number>
                         | <number> . <number>
                         | . <number>
<digit>                  → 0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8| 9
<string>                 → “<word_list>”
<word_list>              → <alphanumeric>
                         | <alphanumeric>   <word_list>
<comment>                → // <word_list> \n
<alphabetic>             → [a-zA-Z]
```

```
<alphanumeric>          → <digit>
                        | <alphabetic>
                        | <digit><alphanumeric>
                        | <alphabetic><alphanumeric>
```

Explanation of the Grammar:

<program> : This is starting variable. This abstraction is for the whole program including functions and comments.

<function_list> : This abstraction is for all functions in the program and commands outside of the functions. In our program, functions should be defined seperately and one can not define a function inside of another function.

<function> : This abstraction is for function definitions. C type langauge rules are followed while its syntax is defined.

<param_list> : This abstraction is for parameter list of function definition. Any number (including zero) of parameters can be defined and they should be seperated by comma.

<stmt_list> : This abstraction is for statements and comments in the function definitions.

<stmt> : It is the building block of statement list. There are several statement types.

<assign_stmt> : It is assignent statment and value of the right side is assigned to identifier in the left.

<expr> : This abstraction stands for expression. A single expression might be single <term>,function call or calculation of other expressions,functions and <term> with low priority operaters such as + and -. Recursion is called from left to handle these operation from left to right. (<expre>) is also an expression to be able to handle priority of phrantheses.

<term> : This abstraction is similar to <expr> but it is added for precedence of operaters. In this abstraction level, there is no low priority operators such as + and - but there is higher priority operator which is &. It will be computed at the bottom levels of parser tree.

<set> : This abstraction states for sets. It is a collection of constant string, number, or other sets. Example: { 1, 2, “Hello world”, { 1, 2}}

<element_list> : It is used to represenet elements of a set. It can be either empty or any number of set elements seperated by comma.

<element> : It is a single set element. It can be either contant strings,numbers,boolean value or other sets or variables.

<decl_stmt> : This abstraction stands for declaration statement. It followes the C type languge syntax.

<identifier_list> : This abstraction is used in the declaration statement and allows us to declare same type of identifiers in a single statement.

<del_stmt> : It is used for free corresponding memory location to given identifier.

<if_stmt> : This abstraction is used for if statements. Each if statement should end with "end" to prevent occuring confussions from nested if statements.

<logic_expr> : These are the building blocks of conditions of if statements. Recursion is called from left to evaluate these expressions from left to right. (<logic_expr>) is also an logic expression to be able to handle priority of phrantheses.

<logic> : These abstractions are building block of logic statements. Result of these abstraction are either true or false. Expressions can be evaluated inside <logic> to be able to use for set relations.

<relation_op> : It is used for relation operations and they are used for comparing sets and elements.

<loop_stmt> :   This abstraction is used for while loop. A while loop starts with the reserved word ‘while’ then the logic expression on parenthesis. After that, statement list inside curly braces.

<return_stmt> : It is used for denoting the return statements of the functions. A return statement is basically starts with ‘return’ followed by an identifier or a primitive type constant. Examples:
return “Hello World”; , return 1256; , return mySet;

<function_call_stmt> : This is for making a function call a statement. It is basically a function call followed by a semicolon. Example: findNumberOfElements(mySet);

<function_call> : This abstraction is used in order to show function calls. It starts with the function name (an identifier) followed by the parameters in parenthesis. Example: findNumberOfElements(mySet)

<input_list> : It is the list of input parameters to call a function. These inputs can be identifiers and constant primitive types.

<write_stmt> : This abstraction is for writing to the console. The reserved word for this functionality is ‘write’. After ‘write’ , the programmer can write an identifier or a primitive type constant. Examples:
write “Hello World”; , write 1256; write mySet;

<read_stmt> : This is for the statements that reads from the console, and assigns the input to the given identifier. Example: read A;

<type> : This abstraction is to denote the primary types of the language, which are bool, string, number and set.

<identifier> : This abstraction states for variable names. A variable should start with a letter either upper or lower case, followed by any combinations of letters and/or digits. Example: mySet2

<string> : This abstraction is for denoting string constants. In order to differentiate strings from identifiers, strings should be inside double quotation marks. And inside of the quotation marks only a <word_list> is accepted, meaning that only upper/lower case letters, digits and space character can be used. Example: “Hello world1 5962kk”

<word_list> : This abstraction stands for one or more <alphanumeric> s separated by space. Example: Hello World 1997

<number> : This abstraction is for various number forms such as integer, float. Examples: 12630, 6, 12.659, .8 etc.

<digit>: This abstraction basically stands for digits.

<alphanumeric> : This abstraction stands for both digits and upper/lower case letters or their various combinations. Examples: 1K4kn9090, Abstraction9783k etc.

<alphabetic> : This abstraction is for uppercase and lowercase letters.

<comment> : This abstraction is for comments. Comments are starting with '//' and ending with a new line. Only upper/lower case letters and digits can be used in comments. Example: // This is 1 comment

Descriptions of Tokens:

Nontrivial tokens:

Comments: "//" followed by space seperated alphanumeric characters until new line stands for comment token in our program. We decided to use "//" as begining point to follow C type language's syntax. This is the only way of having commands and it improves the overall simplicity of our language. Therefore it affects readibility, writability and reliability positively.

Identifiers: Identifiers starts with alphabetic character and followed by zero or more alphanumberic character. However, special words are not allowed to use in identifiers. This improves our language's syntax design since it prevents possible confusions. Having better syntax design leads to an improvment in terms of readibility,writability and reliability.

Reserved Words: Reserved words are names of primitive types (string, set, number bool) , 'while', 'read', 'if', 'delete', 'end', 'else', 'write', and 'return'. We tried to choose those words in a way that it will be easy to understand when reading and easy to memorize in order to write. Reserved words can not be used as identifiers. This increases the readability and reliaility of the language because it prevents the reader from confusions.

Literals: In this language literals cannot include special characters other than that upper/lower case letters and digits. Literals for bool are 'true' and 'false'. And these literals cannot be used as identifiers. Literals for 'number' are all number in the forms of integer or float. These also can not be used as identifiers. Literals for 'string' is ay <word_list> between double quotation marks. Again, strings can not be used as identifiers. Literals for 'set' is any <element_list> between curly parentheses. These also cannot be used as identifiers. Literals not being used as identifiers increases the readability and reliability of the language.

Evaluation of the Language:

Readability:

We can analyze readability of this language in terms of operators and their meanings. This language is mainly interested in sets and set operations. It is a major requirement to have union, intersection, difference operations. '-' operator stands for difference operation, which is intuitively correct. However, '+' and '&' operators stands for union and intersection, respectively. Understanding the meaning of these operators can be confusing, because they are different than their corresponding mathematical notations. This situation decreases the readability of our language.

Other point about readability is curly braces. In this language matching curly braces are used while denoting sets, functions and loops. Therefore, it might be confusing to determine which '}' ends which compound statement group. However, 'if' statements are ended with the reserved word 'end', making it easy to understand where the 'if' statement is done. Using curly braces for different types of structures diminishes the readability of our language, while using 'end' for only if statements is positive in terms of readability.

In this language, the programmer can not used reserved words as identifiers. Reserved words are primitive types, 'while', 'read', 'if', 'delete' etc. This feature increases the readability of our language, because it prevents the reader from confusion.

Writability:

We can analyze writability of this language on terms of orthogonality. In this language, any combination of number/string/set + number/string/set is valid, which makes '+' operator and data types orthogonal. This allows the programmer to have extended functionality of addition operation.

In this language, curly braces are used in order to denote the statements inside functions, loops and denote the set elements. However, curly braces are not used for if statements ('end' is used instead). Using different techniques for the same purpose might be confusing when it comes to writing. Therefore, while this situation makes the readability of this language increase (explained before), the writability of the language decreases because of that.

Another point about writability is powerful operators. The '?' operator saves the programmer from dealing with longer expressions. '?' is used for checking if an element exists in a given set, such as A ? { 1, 2, 3} . Without the '?' operator the programmer should have written it as {A} & { 1, 2, 3} != { } . This situation improves expressivity, therefore, writability.

Reliability:

We can analyze reliability of this language in terms of aliasing. Aliasing is not possible in our language, which means that there can not be more than one name in order to reach a specific memory cell. Therefore, this prevents the programmer from changing a variable while the intention was to change some other variable. This situation increases the reliability of our language.

Another way to analyze the reliability of a language is to look at type checking and exception handling. Since our language is currently in grammar and lex stage, we can not analyze our language in term of type checking and exception handling.