CS315 - Programming Languages
Project 2
Team 05

İrem Vesile Aydın 21902914

Zeynep Derin Dedeler 21902896

Ece Kahraman 21801879

1. BNF Description

<program> ::= START <code> END

<code> ::= <statement_list> | <statement_list>? <main> <statement_list>?

<statement_list> ::= <statement> | <statement> <statement_list>

<main> ::= MAIN_FUNC LP RP <block>

<statement> ::= <conditional_statement>
| <assignment_statement> SC
| <comment_statement>
| <del_statement> SC
| <loop_statement>
| <declaration_statement> SC
| <return_statement> SC
| <func_call_statement> SC

<conditional_statement> ::=
IF LP (<rel_expression>| <func_call> | <var_ident>) RP <block> ELSE <block>
| IF LP (<rel_expression>| <func_call> | <var_ident>) RP <block>
| IF LP (<rel_expression>| <func_call> | <var_ident>) RP <block>
(ELSE_IF LP <block> RP)+(ELSE <block>)?

<assignment_statement> ::= <set_ident> ASSIGN_OP <set_expression>
| <var_ident> ASSIGN_OP (<element> | <string> | <bool>)
| <var_ident> ASSIGN_OP <int_expression_1>
| (<set_ident> | <var_ident) ASSIGN_OP <func_call>

<comment_statement> ::= COMMENT {<cap_letter> | <letter> | <digit> | <symbols> | <empty> | <white_space> } ENDLINE

<del_statement> ::= DEL <set_ident>

<loop_statement> ::= WHILE LP <rel_expression> RP <block>
| WHILE LP <bool> RP <block>
| WHILE LP <func_call> RP <block>
| DO <block> WHILE LP <rel_expression> RP

```
                        | DO <block> WHILE LP  <func_call>  RP
                        | DO <block> WHILE LP  <bool>  RP


<declaration_statement> ::= <set_declaration>
                                | <func_declaration>
                                | <element_declaration>
                                | <string_declaration>
                                | <int_declaration>
                                | <bool_declaration>

<return_statement>::= RETURN  (<type_ident> | <type_init>)

<func_call_statement> ::= <func_call>

<block> ::= LB <statement_list>? RB

<set_declaration>::= SET_KEYWORD  (<set_ident> ASSIGN_OP  <set_expression>
                                        | <set_ident> ASSIGN_OP <set_init>
                                        | <set_ident> ASSIGN_OP <func_call>)

<func_declaration> ::=
                FUNC_KEYWORD <func_ident> LP <typed_argument_list> RP <block>

<element_declaration> ::= ELEMENT_KEYWORD  <var_ident> (ASSIGN_OP
                                (<func_call> | <element>) )?

<string_declaration> ::= STRING_KEYWORD <var_ident> (ASSIGN_OP
                                (<func_call> | <string>) )?

<bool_declaration> ::= BOOLEAN_KEYWORD <var_ident> (ASSIGN_OP
                                (<func_call> | <bool>) )?

<int_declaration> ::= INT_KEYWORD <var_ident> ( ASSIGN_OP <func_call>
                                                        | ASSIGN_OP <int_expression_1>)?

<int_expression_1> ::=  (<var_ident> | <int>)
                        | <int_expression_1> <low_prec_int_op> <int_expression_2>
                        | <int_expression_2>
```

<int_expression_2> ::= (<var_ident> | <int>)
                        | <int_expression_2> <high_prec_int_op> (<var_ident> | <int>)

<set_expression> ::= <set_ident> | <set_expression> <set_operator> <set_ident>

<rel_expression> ::= <set_ident> <rel_operator> <set_ident>
                        | <var_ident> <int_rel_operator> (<var_ident> | <int>)
                        | <int> <int_rel_operator> <var_ident>
                        | <var_ident> <bool_rel_operator> <bool>

<argument_list>::=   <empty>
                        | <type_init> | <type_ident>
                        | <type_ident> COMMA <argument_list>
                        | <type_init> COMMA <argument_list>

<typed_argument_list>::=   <empty>
                        | <type_keyword> <type_ident>
                        | <type_keyword> <type_ident> COMMA <typed_argument_list>

<element_list> ::= <element> | <element> COMMA <element_list>

<set_ident> ::= <cap_letter> {<letter>|<digit>}

<var_ident> ::= <letter>{<cap_letter> | <letter> | <digit>}

<set_init> ::= LB [<element_list>] RB

<element> ::= LQ{<letter>|<digit>|<symbols>|<cap_letter>}RQ

<string>::= LDQ{<cap_letter> | <letter> | <digit>|<symbols>}RDQ

<int> ::= (-)?(<digit>)+

<primitive_func> ::= <add> | <remove> | <print> | <print_file> | <is_empty> | <get_size>
                | <contains> | <pop> | <read_file> | <read>

<add> ::= ADD_FUNC LP <set_ident> COMMA (<element> | <var_ident) RP

<remove> ::= REMOVE_FUNC LP <set_ident> COMMA <element> RP

<print> ::= PRINT_FUNC LP (<set_ident> | <var_ident> | <string>) RP

<print_file> ::= PRINTFILE_FUNC LP <string> COMMA (<set_ident> | <var_ident) RP

<is_empty> ::= ISEMPTY_FUNC LP <set_ident> RP

<get_size> ::= GETSIZE_FUNC LP <set_ident> RP

<contains> ::= CONTAINS_FUNC LP <set_ident> COMMA (<element> | <var_ident>)

<pop> ::= POP_FUNC LP <set_ident> RP

<read_file> ::= READFILE_FUNC LP <string> RP

<read> ::= READ_FUNC LP (<type_ident> | <string> | <int> | <element>)

<func_call> ::= <func_ident> LP (<argument_list> | <rel_expression> | <func_call>) RP
| <primitive_func>

<func_ident>::= UNDERSCORE<letter> {<letter>|<digit>}
| UNDERSCORE<primitive_func>

<set_rel_operator> ::= SUBSET_OF | SUPERSET_OF | EQUAL | NOT_EQUAL

<int_rel_operator> ::= LESS_THAN | GREATER_THAN | LESS_OR_EQUAL |
GREAT_OR_EQUAL | EQUAL | NOT_EQUAL

<bool_rel_operator> ::= EQUAL | NOT_EQUAL

<set_operator> ::= OR_OP | AND_OP | SUB_OP

<low_prec_int_op> ::= ADDITION_OP | SUBTRACTION_OP

<high_prec_int_op> ::= MULTIPLICATION_OP | DIVISION_OP

<type_ident> ::= <var_ident> | <set_ident>

<type_init> ::= <set_init> | <element> | <string> | <int> | <bool>

<type_keyword> ::= STRING_KEYWORD | SET_KEYWORD | ELEMENT_KEYWORD | INT_KEYWORD | BOOLEAN_KEYWORD

<bool> ::= TRUE | FALSE

2. General Description

YASS (Yet Another Statement Set) is a programming language targeting mathematical sets and related operations. We specified a data type for sets, which are a list of elements between curly brackets, and a data type for elements, which are any characters written between single quotes. The program is written inside the main function and each line is ended with a semicolon. We have defined some primitive functions and the user can define their own functions. Additional to the functions, the user also use loops, relations between integers and sets, declarations, conditional statements, and arithmetic expressions.

Users can use loop statements by using “while” keyword and following it with relational expressions between brackets. Alternatively, they can also use the do-while loop with the keywords “do” and “while” to have a loop that iterates at least one. Users can combine these inside a function declaration or can directly call primitive functions.

3. Explanations for Each Language Construct

<program> It stands for written the program. It limits the user to only write the code between start and end commands.

<code> It stands for actual program body consits of <statement_list> and optional one main function.

<main> It stands for the main function of the program. The main function’s code is between curly brackets. Inside the main function, the user can still declare functions.

<statement_list> This one stands for one or more statements that build the main part of the program.

<statement> This one stands for a list of statement types in our language. Our language has eight types of statements, which are <conditional_statement>, <assignment_statement>, <comment_statement>, <del_statement>, <loop_statement>, <declaration_statement>, <return_statement>, and <func_call_statement>. Only comments, loops, and conditionals do not require a semicolon at the end of their lines.

<conditional_statement> This one represents out if-elif-else structure. if and elif parameters can be a relational expression between sets, integers, or can check the boolean parameters and variables.

<assignment_statement> This one is to assign values to an already existing set or variable. A set variable can be assigned a set expression, a new set value or a return value of a function, or a non-set variable can be assigned a value, an arithmetic expression or a return value of a function.

<comment_statement> This statement is not executable and allows the user to comments and takes note during to code to explain it.

<del_statement> This statement represents the deallocation of set variables. We wanted to add this to give the user some writing flexibility.

<return_statement> This is to return values when they are needed. It uses the keyword “return” with the name of set or non-set variables.

<loop_statement> This represents the possible loop structures. We implemented a while loop and a do-while loop. Both of them can take relational expressions, function calls and boolean values as their parameters.

<declaration_statement> This is the combination of all declaration statements, which are <set_declaration>, <func_declaration>, <element_declaration>, <string_declaration>, <int_declaration>, <bool_declaration>. All of them uses their respective keywords to declare new set and non-set variables, and user defined functions.

<block> This construct represents the scope of the block which is closed by curly brackets. It can also be an empty scope.

<empty> An empty space.

<set_declaration> This one combines the keyword “set” with a set identifier, whose first letter has to be uppercase. In this part, the user can either leave it as just a declaration, or assign it a value by some set operation expressions, an element list, or the name of another set.

<func_declaration> To declare a function, we use the keyword “function” for type declaration, what the function returns is not important in our language. The function names need to start with an underscore and their first letter must be lowercase. They can have multiple variables as parameters as long as their types are declared inside the signature.

<element_declaration> We have imagined an element variable as different from other types of variables. It uses the “element” keyword followed by an identifier. An element has to be enclosed by single quotes like chars, and inside the quotes, the user can write whatever they like. There is no difference between a character and an integer if it is declared as an element. They are only declared to fill existing sets.

<int_declaration> Following the keyword “int”, an int variable can be left as a declaration, can be assigned a value, another int variable, or some arithmetic expression.

<string_declaration> Using the keyword “string”, a string variable can either be just declared, or be assigned a string inside double-quotes.

<bool_declaration> Using the keyword “bool”, a bool variable can only be assigned a boolean value, “true” or “false”.

<int_expression_1> This construct can take an int variable of an int value. This one specifically calls the low precedent arithmetic operators, ADDITION_OP and SUBTRACTION_OP. It also calls <int_expression_2>, which calls the high precedent arithmetic operators MULTIPLICATION_OP and DIVISION_OP.

<int_expression_2> This one calls for the high precedent arithmetic operators for the int expression, MULTIPLICATION_OP and DIVISION_OP.

<set_expression> This construct is used to assign set variables to another existing set identifier, or to an expression of which set operators OR_OP (union), AND_OP (intersection), and SUB_OP (difference) are used. There is no specific precedence of set operations, every expression is executed from left to right.

<rel_expression> This construct groups all kinds of relational expressions we have in our language. It calls <rel_operator> to check the relations between two sets, calls <int_rel_operator> to compare two int values, calls <bool_rel_operator> to compare one bool variable to TRUE or FALSE.
<argument_list> This construct is for the parameter lists for the already declared functions. It can either be a list of variable identifiers, or a list of values. Either way, it takes no data type keywords. It is called when a function call happens.

<typed_argument_list> The same idea as <argument_list>, but this time it can only be a list of variable identifiers with related data type keywords. This is used when a function declaration/initialization is happening.

<primitive_func> This groups the primitive functions which are implemented in our language. We designed some primitive functions for our language YASS.

- ***<add>*** This function adds an element to a set and returns true if the operation is successful or returns false.
  _add(Set1, ‘el’);

- ***<remove>*** This function removes an element from a set and returns true if the operation is successful or returns false.
  _remove(Primeset, ‘12’);

- ***<print>*** This function prints four primitive types to the environment console. It can print strings, elements, integers and sets. It can take either their identifiers or direct values.
  _print(“Testing the string 12.3”);
  _print(el12);
  _print(Set21);

- ***<printFile>***This function takes a file destination in the string form and prints the content of the file to the environment console.
  _printFile(“C:\Users\zeynep\Documents\zdd-Dersler\CS315”);

- ***<isEmpty>*** This function returns true if the set is empty. Otherwise, it returns false.
  _isEmpty(Set8);

- ***<getSize>*** This function gets a set as a parameter and returns the number of elements it contains in integer type.
  _getSize(Set3);

- ***<contains>*** This function return true if the set contains the element. Otherwise, it returns false.
  _contains(Setsm1, ‘sm1’);

- ***<pop>*** This function pops the last inserted element from a set and returns true. If the set is empty it returns false.
  _pop(Set21);

- ***<readFile>*** This function takes a file destionation and reads its content and returns it in a string form to the program
  _readFile(“C:\Users\zeynep\Documents\zdd-Dersler\CS315”);

- ***<read>*** This function reads input from the user. Element, integer and string can be passed as arguments to the function. It reads from the keyboard and assigns the value to the passed argument.
  _read(S1);
  _read(elm);

<set_operator> This one calls OR_OP (union), AND_OP (intersection), or SUB_OP (difference).

<set_rel_operator> This one calls SUBSET_OF, SUPERSET_OF, EQUAL, NOT_EQUAL for set variables.

<int_rel_operator> Calls LESS_THAN, GREATER_THAN, LESS_OR_EQUAL, GREAT_OR_EQUAL, EQUAL, NOT_EQUAL for int values.

<int_operator_1> Calls the low precedence arithmetic operators for int values, ADDITION_OP, SUBTRACTION_OP

<int_operator_2> Calls the high precedence arithmetic operators for int values, MULTIPLICATION_OP, DIVISION_OP

<bool_rel_operator> Calls for EQUAL and NOT_EQUAL for checking boolean values.

4. Nonterminals

START: indicates the start of the program which is represented by “start”

END: indicates the end of the program which is represented by “end”

LP: left parenthesis

RP: right parenthesis

LB: left bracket

RB: right bracket

LQ: left quote

RQ: right quote

LDQ: left double quote

LRQ: right double quote

SC: semicolon

COMMA: comma

UNDERSCORE: underscore

COMMENT: two slashes

ENDLINE: endline character

IF: “if” keyword to start the if conditional

ELSE_IF: “elif” keyword to start the else-if conditional

ELSE: “else” keyword to start the else block

WHILE: “while” keyword to start the while loop

DO: “do” keyword to start the do-while loop

DEL: “del” keyword to delete sets

RETURN: “return” keyword to return values

MAIN_FUNC: “_main” function keyword

ADD_FUNC: “_add” primitive function keyword

REMOVE_FUNC: “_remove” primitive function keyword

PRINT_FUNC: “_print” primitive function keyword

PRINTFILE_FUNC: “_printFile” primitive function keyword

ISEMPTY_FUNC: “_isEmpty” primitive function keyword

GETSIZE_FUNC: “_getSize” primitive function keyword

CONTAINS_FUNC: “_contains” primitive function keyword

POP_FUNC: “_pop” primitive function keyword

READFILE_FUNC: “_readFile” primitive function keyword

READ_FUNC: “_read” primitive function keyword

SET_KEYWORD: “set” keyword to declare sets

ELEMENT_KEYWORD: “element” keyword to declare elements

STRING_KEYWORD: “string” keyword to declare strings

INT_KEYWORD: “int” keyword to declare integers

BOOLEAN_KEYWORD: “bool” keyword to declare booleans

ASSIGN_OP: assignment operator which represents “=”

LESS_THAN: less than operator which represents “<”

GREATER_THAN: greater than operator which represents “>”

LESS_OR_EQUAL: less or equal operator “<=”

GREAT_OR_EQUAL: greater or equal operator “>=”

EQUAL: equality operator which represents “==”

NOT_EQUAL: not equal operator which represents “!=”

SUBSET_OF: operator for determining subsets which represents “<<”

SUPERSET_OF: operator for determining supersets which represents “>>”

OR_OP: operator for set unions which represents “or”

AND_OP: operator for set intersections which represents “and”

SUB_OP: operator for set differences which represents “sub”

ADDITION_OP: “+” operator for integers

SUBTRACTION_OP: “-” operator for integers

MULTIPLICATION_OP: “*” operator for integers

DIVISION_OP: “/” operator for integers

TRUE: represents the boolean value “true”

FALSE: represents the boolean value “false”

5. Description of Nontrivial Tokens

Comments (//): Anything written after a double slash is interpreted as a comment. The comment takes the whole line and unlike other statements, they do not end with semicolons. Double slash is easy to write and remember since many languages use it to make a comment. It also provides a readable syntax since double slash is not used anywhere in the program. We decided to describe one way to make comments to make the program readable.

Identifiers: Our language (YASS) has two types of identifiers.Set identifiers and variable identifiers.

Set Identifiers are used for only set objects. Set identifiers' existence allows users to differentiate set objects from other types of variables in our language. Set identifiers are defined as starting with a capital letter and continuing with lower case letters or digits. This type of identifier helps with the readability and usability of the code because the user needs to track their existing set objects to delete them later and a different identifier for sets helps users to notice them and avoid confusing sets variables to a different type of variable.

Variable identifiers are used for elements, integers, strings and booleans. Variable identifiers are defined as starting with a lower case letter and continuing with lower case letters or digits.

These identifiers generally help with the writebility and readability of the code. The existence of identifiers allows users to give their variables meaningful names thus making it easy to remember their purpose and values

Literals: YASS has four types of literal: int, bool, element and string. These literals cover all the functionality the language aims for and provide writability to it. Users can define an int by using the int keyword and initialize it using an integer value. Strings are created using the string keyword and initialized by characters enclosed by double quotation marks. Elements are created by using the element keyword and initialized by characters enclosed by single quotation marks. Booleans are created by using the bool keyword and initialized by true or false.

We decided not to have double literal in the YASS language since it has no functionality in the language. By eliminating unnecessary literals, we aim to provide readability and writability to the language.