Bilkent University

Department of Computer Science

CS 315 Project 1

Section 1

Team 15

Programming Language Name: AYVA++

Project Report

Group Members:

- Mehmet Alper ÇETİN - 21902324
- Recep UYSAL - 21803637
- Alperen CAN - 21601740

Contents

1. Complete BNF Description

1.1 Constants and Types

- Ayva++ has the following types integer, double, boolean, member and set.

```
<assign_op> ::= <<

<LP> ::= (

<RP> ::= )

<LB> ::= {

<RB> ::= }

<low_char> ::=  [a-z]

<up_char> ::= [A-Z]

<digit> ::= [0-9]

<comment_sign> ::= #

<new_word> ::= new

<end_stmt> ::= :

<integer> ::= <sign>?<number>

<double> ::= <sign>?(<number><dot><number>)

<number> ::= 0 | <nonzero_digit><digit>*

<nonzero_digit> ::= [1-9]

<sign> ::= + | -

<arith_sign> ::= + | - | * | /

<identifier> ::= <low_char>

                 | <identifier><low_char>

                 | <identifier><digit>

<boolean> ::= true | false

<space> ::=

<dot> ::= .

<comma> ::= ,

<comparison_sign> ::=  < | <= | > | >= | == | !=

<logical_operator> ::= && |  ||

<set_identifier> ::= <up_char>
```

```
                  | <identifier><up_char>
                  | <identifier><digit>
<file_name> ::= <string>
<member> = <string> | <integer> | <identifier> | <double>
```

1.2 Program

```
<program> ::= <main>
<main> ::= <LP><RP>begin<statements>end
<statements> ::= <statement>
                   |   <statements><statement>
<statement> ::= <expr><end_stmt> | <loop> | <conditional> | <function> |
                   | <comment>
<words> ::= (<up_char>| <low_char> | <digit> | <space>)+
<comment> ::= <comment_sign><words>?<comment_sign>
<expr> ::= <int_exp> |  <boolean_exp> | <set_exp> | <function_call_exp>
           | <print_exp> | <double_exp>
```

1.3 Conditionals

```
<conditional> ::= <if_stmt><else_stmt>?
<if_stmt> ::= if <LP><logicals><RP> begin <statements> end
<else_stmt> ::= else begin <statements> end
<logical> ::= <identifier><comparison_sign><integer>
              | <identifier><comparison_sign><identifier>
              | <integer><comparison_sign><identifier>
              | <set_relations>
<logicals> ::= <logical>
              | <logicals><logical_operator><logical>
```

1.4 Loops

```
<loop> ::= <while> | <for> | <do_while>
```

<while> ::= while <LP><logicals><RP> begin <statements> end

<for> ::= for <LP><ident_exp><RP> begin <statements> end

<do_while> ::= do begin <statements> end while <LP><logicals><RP><end_stmt>

1.5 Functions

```
<function> ::= func <function_call> begin <statements> return
                    <return_type> end
<return_type> ::= <argument>
<argument> ::= <identifier> | <integer> | <boolean> | <set_identifier>
<arguments> ::= <argument>
                     |   <arguments><comma><argument>
<function_call> ::= <identifier><LP><arguments><RP>
```

1.6 Expressions

```
<ident_exp> ::= <identifier><assign_op><integer><end_stmt><logicals><end_stmt>
                       <integer>
<int_exp> ::= <identifier><assign_op><integer>
                 |  <identifier><assign_op><identifier>
                 |   <int_exp><arith_sign><integer>
                 |  <int_exp><arith_sign><identifier>
<double_exp> ::= <identifier><assign_op><double>
                 |   <double_exp><arith_sign>(<integer>|<double)
<boolean_exp> ::= <identifier><assign_op><boolean> | <logicals> | <set_relations>
<fuction_call_exp> ::= <identifier><assign_op><function_call>
                          | <set_identifier><assign_op><function_call>
<print_exp> ::= print <LP><string><RP>
<string> ::= “<words>”
```

1.7 Sets

<set_methods> ::= <set_delete> | <set_remove> | <set_add> | <set_cardinality>

```
                          | <set_read> | <set_print>
<set_operations> ::= <set_union> | <set_intersection> | <set_difference> | <set_init>
<set_relations> ::= <is_empty> | <is_subset> | <is_superset>
<set_delete> ::= <set_identifier><dot> delete <LP><RP>
<set_remove> ::= <set_identifier><dot> remove <LP><member><RP>
<set_add> ::= <set_identifier><dot> add <LP><member><RP>
<set_cardinality> ::= <set_identifier><dot> getCardinality<LP><RP>
<set_read> ::=  <set_identifier><dot> read<LP><file_name>?<RP>
<set_print> ::=  <set_identifier><dot> print<LP><file_name>?<RP>
<set_union> ::= <set_identifier><dot> union <LP><set_identifier><RP>
<set_intersection> ::= <set_identifier><dot> intersection <LP><set_identifier><RP>
<set_difference> ::= <set_identifier><dot> difference <LP><set_identifier><RP>
<is_subset> ::= <set_identifier><dot> isSubset <LP><set_identifier><RP>
<is_superset> ::= <set_identifier><dot> isSuperSet <LP><set_identifier><RP>
<is_empty> ::= <set_identifier><dot> isEmpty <LP><RP>
<set_init> ::= new Set
                  | new Set <LB><arguments><RB>
<set_exp> ::= <set_identifier><assign_op><set_operations>
```

2. Explanation of AYVA++ Constructs

2.1 Nonterminals

- **<assign_op> ::= <<**

  In our language, initializations are done by “<<” operator.

- **<comment_sign> ::= #**

  For adding comment, users must use “#” both in the beginning and end of a sentence.

- **<new_word> ::= new**

  New keyword is used for set initialization.

- **<end_stmt> ::= :**

  In Ayva++, every statement should be end by “:”.

- **<number> ::= 0 | <nonzero_digit><digit>***

  In our language, numbers start by zeros are not acceptable. For instance 0016, 03, etc. are undefined

- **<identifier> ::= <low_char> | <identifier><low_char> | <identifier><digit>**

  In Ayva++, identifiers are consist of only lower case characters and digits.

- **<set_identifier> ::= <up_char> | <identifier><up_char> | <identifier><digit>**

  Set names are consist of only upper case characters and digits in our language.

2.2 Program Structure

- **<program> ::= <main>**

The program is constructed on “main” components, which enables to start the program in AYVA++.

- **<main> ::= <LP><RP>begin<statements>end**

Program is started by main including statements.

- **<statements> ::= <statement>**

  | <statements><statement>

Statements contain either one statement or more.

- **<statement> ::= <expr><end_stmt> | <loop> | <conditional> | <function> |**

  | <comment>

Statement includes expressions, loops, conditional statements, functions and comment lines.

- **<comment> ::= <comment_sign><words>?<comment_sign>**

Comment lines starts and ends with a comment sign. Between comment signs, words might or might not occur. It enables users to write words to explain their code for clarity.

- **<words> ::= (<up_char> | <low_char> | <digit> | <space>)+**

Words consist of one or several characters, digits or spaces.

- **<expr> ::= <int_exp> | <boolean_exp> | <set_exp> | <function_call_exp>**

  | <print_exp> | <double_exp>

An expression is either an integer expression, a boolean expression, a set expression, a function call expression, a print expression, or a double expression. Expressions will be explained in detail in Section 2.5

2.3 Conditionals

- **<conditional> ::= <if_stmt><else_stmt>?**

A conditional statement can be a single or more if statements continued with/without an else statement

- **<if_stmt> ::= if <LP><logicals><RP> begin <statements> end**

An If statement starts with “if” keyword and it is continued with a logical part in parentheses and it is followed by the statements part, which is placed between “begin” and “end” keywords.

- **<else_stmt> ::= else begin <statements> end**

In “else” statement, “else” keyword is followed by the statements part which is located between “begin” and “end” keywords.

- **<logical> ::= <identifier><comparison_sign><integer>**

  | <identifier><comparison_sign><identifier>

  | <integer><comparison_sign><identifier>

  | <set_relations>

“Logical” is used for making comparisons between identifiers, or making comparisons between an identifier and an integer. i.e 5 < int1. See Section 1.1 for comparison signs.

- **<logicals> ::= <logical>**

  | <logicals><logical_operator><logical>

Logicals consist of one or more logical expressions. There must be a logical operator sign between them, i.e. logical && logical. See Section 1.1 for logical operator representations.

2.4 Loops

- **<loop> ::= <while> | <for> | <do_while>**

In our language defined loops are “do while”, “for”, and “while” loops. It provides continuous repetition of the statements.

- **<while> ::= while <LP><logicals><RP> begin <statements> end**

In “while” loop, the statements which are inside of “begin” and “end” keywords will be repeated based on the given logical expression(s).

- **<for> ::= for <LP><ident_exp><RP> begin <statements> end**

In “for” loop, after the “for” keyword, an identify expression will be given inside the parentheses, which also includes logical expressions (See Section 2.5). The statements will be given inside the “begin” and “end” keywords.

- **<do_while> ::= do begin <statements> end while <LP><logicals><RP><end_stmt>**

In “do while” loop, the statements inside the “begin” and “end” keywords will be executed at least once. The repetition will be controlled by the logical expressions inside the parentheses at the end.

2.5 Functions

Functions of AYVA++ are implemented in our main’s body, and they are called from main. User has to type return even if it returns nothing, in order to stop the function.

- **<function> ::= func <function_call> begin <statements> return**

  <return_type> end

Functions start with the “func” keyword and continue with the function call part. The function body starts with “begin” keyword, followed by statements, and ends with “end” keyword.

- **<return_type> ::= <argument>**

Return type equals to argument.

- **<argument> ::= <identifier> | <integer> | <boolean> | <set_identifier>**

An argument can be an identifier, an integer, a boolean or name of a set.

- **<arguments> ::= <argument>**

  | <arguments><comma><argument>

Arguments consist of a single argument or more arguments separated by commas.

- **<function_call> ::= <identifier><LP><arguments><RP>**

In function call, there is an identifier followed by arguments which are given between parentheses. The identifier part represents the function name.

2.6 Expressions

- **<ident_exp> ::= <identifier><assign_op><integer><end_stmt><logicals>**

  <end_stmt> <integer>

Identify expression is used in "for" loops, containing the expression inside the parentheses: an identifier, assignment operator, integer, followed by a logical expression after separated by the end statement ":"

- **<int_exp> ::= <identifier><assign_op><integer>**

  | <identifier><assign_op><identifier>

  | <int_exp><arith_sign><integer>

  | <int_exp><arith_sign><identifier>

Integer expression includes assigning an identifier to an integer, or assigning an identifier to arithmetic operations of several integers or identifiers.

- **<double_exp> ::= <identifier><assign_op><double>**
  | <double_exp><arith_sign>(<integer>|<double)

Double expression includes assigning an identifier to an double, or assigning an identifier to arithmetic operations of several doubles or integers.

- **<boolean_exp> ::= <identifier><assign_op><boolean> | <logicals>**

  | <set_relations>

Boolean expressions are the expressions which have a boolean value true or false. It might be an identifier followed by the assignment operator and boolean value, or logical expressions which compare identifiers and integers (see Section 2.2), or set relations (see Section 2.6).

- **<fuction_call_exp> ::= <identifier><assign_op><function_call>**

  | <set_identifier><assign_op><function_call>

Function call expression is used for assigning an identifier or a set to function's return. Identifier can get the value of an integer or a boolean or a set or an identifier.

- **<print_exp> ::= print <LP><string><RP>**

Print expression consists of "print" keyword followed by a string inside of parentheses.

- **<set_exp> ::= <set_identifier><assign_op><set_operations>**

Set expression initializes a set by doing set operations. For clarity, see Section 2.6 for set operations.

- **<string> ::= "<words>"**

String will be used for print expression as well as showing set elements. See <words> in Section 2.1 for clarity.

2.7 Sets

- **<file_name> ::= <string>**

Name of the file is a string.

- **<member> = <string> | <integer> | <identifier>**

Members of the sets can be strings, integers or identifiers.

- **<set_methods> ::= <set_delete> | <set_remove> | <set_add>**

  | <set_cardinality> | <set_read> | <set_print>

The methods related with sets involve functions such as deleting a set, removing a set element, adding an element to set, getting the cardinality of a set, reading and printing a set.

- **<set_delete> ::= <set_identifier><dot> delete <LP><RP>**

Set delete method deletes the elements of the given set.

- **<set_remove> ::= <set_identifier><dot> remove <LP><member><RP>**

Set remove method removes an element from a set. The element to be removed is given in parentheses.

- **<set_add> ::= <set_identifier><dot> add <LP><member><RP>**

Set add method adds an element to a set. The element to be added is given in parentheses.

- **<set_cardinality> ::= <set_identifier><dot> getCardinality<LP><RP>**

Set cardinality method returns the number of elements of the given set.

- **<set_read> ::= <set_identifier><dot> read<LP><file_name>?<RP>**

Set read method reads a set either from a file or from the console. If no file name is written between parentheses, the set will be read from the console.

- **<set_print> ::= <set_identifier><dot> print<LP><file_name>?<RP>**

Set print method prints the elements of the given set. Also, it saves the content of a set to a file, if a file name is given into parentheses.

- **<set_operations> ::= <set_union> | <set_intersection> | <set_difference>**

  | <set_init>

Set operations consist of union, intersection and difference operations as well as initializing a set.

- **<set_union> ::= <set_identifier><dot> union <LP><set_identifier><RP>**

Set intersection finds the all elements that are in the first set or the second set. It returns element(s) of a set.

- **<set_intersection> ::= <set_identifier><dot> intersection <LP><set_identifier><RP>**

Set intersection finds the elements which are in the both first and second set. It returns element(s) of a set.

- **<set_difference> ::= <set_identifier><dot> difference <LP><set_identifier><RP>**

Set difference finds the elements which are in the first set but not in the second set. It returns element(s) of a set.

- **<set_init> ::= new Set**

  | new Set <LB><arguments><RB>

Set init is a keyword for creating a new set. You can create a empty set or give arguments.

- **<set_relations> ::= <is_empty> | <is_subset> | <is_superset>**

Set relations include functions controlling whether a set is a subset of another set, whether a set is empty or not, or whether a set is superset of another set

- **<is_empty> ::= <set_identifier><dot> isEmpty <LP><RP>**

IsEmpty method checks whether a set is empty or not. It returns a boolean.

- **<is_subset> ::= <set_identifier><dot> isSubset <LP><set_identifier><RP>**

IsSubset method checks whether the first set whose name is placed before the dot is subset of the second set whose name is placed between parentheses. It returns a boolean.

- **<is_superset> ::= <set_identifier><dot> isSuperSet <LP><set_identifier><RP>**

IsSuperset method checks whether the first set is superset of the second set which is placed between parentheses. It returns a boolean.

3. Descriptions of Nontrivial Tokens

- **MAIN:** Token for starting the program.
- **END_STMT:** Token for end statement sign.
- **IF:** Token for conditional statements.
- **ELSE:** Token for conditional statements.
- **FUNC_DEC:** Token for function declaration.
- **INT:** Token for integer types.
- **DOUBLE:** Token for double types.
- **NEW:** Token for set declaration.
- **BOOLEAN:** Token for boolean types.
- **LOGICAL_OP:** Token for logical operations.
- **IDENTIFIER:** Token for identifiers.
- **WHILE:** Token for detecting while loops.
- **FOR:** Token for detecting for loops.
- **RETURN:** Token for return operator.
- **COMMENT:** Token for finding comments.
- **SET_İDENTİFİER:** Token for set names.
- **DELETE:** Token for deleting a set.
- **REMOVE:** Token for removing an element from a set.
- **ADD:** Token for adding an element to a set.
- **CARDINALITY:** Token for getting the cardinality of a set.
- **READ:** Token for reading set elements from a file/console.
- **PRINT:** Token for printing set elements.
- **UNION:** Token for finding union of two sets.
- **INTERSECTION:** Token for finding intersection of two sets.
- **DIFFERENCE:** Token for finding difference of two sets.
- **SUBSET:** Token for checking whether a set is a subset of another set.
- **SUPERSET:** Token for checking whether a set is a superset of another set.
- **EMPTY:** Token for checking whether a set is empty.

4. Evaluation of AYVA++

4.1 Readability

Readability refers to the ease with which programs can be read and understood. The evaluation criteria of readability involves overall simplicity, orthogonality, data types and syntax considerations.

a. Overall Simplicity

In terms of feature multiplicity, more than one way exists to perform a specific operation in AYVA++. For instance, repeating a set of instructions can be done by using different looping constructs such as do while, for and while loops. Therefore, it causes a decreasement in readability of the program by affecting the simplicity of the language.

b. Orthogonality

Functions of AYVA++ can return many types such as sets, integers and booleans. Therefore, it enhances the orthogonality of the language as well as the readability of it.

c. Data Types

AYVA++ does not include unnecessary data types. All data types such as integers, sets and chars are essential for our language, which is designed for finite sets. Therefore, AYVA++ contains sufficient predefined data types, which is also beneficial for the readability of the language.

d. Syntax Considerations

In order to make the syntax clear, we are using matching keywords for the body of the conditional statements and loops of AYVA++, such as “begin” and “end”. However, excessive and nested use of these statements can lessen the readability of AYVA++.

4.2 Writability

Writability refers to the ease with which a language can be used to create programs. The evaluation criteria of writability involves simplicity and orthogonality, support for abstraction and expressivity.

a. Simplicity and Orthogonality

AYVA++ developers will not face difficulty in terms of the rules of the language and usage of constructors, since all constructs are essential, and designed specifically for the aim of the language, which is doing set operations. AYVA++ does not include an excessive number of constructs. Therefore, it provides increased writability.

b. Support for Abstraction

Our language allows its developers to define functions. Since a function is essentially an abstraction, it increases the writability of AYVA++.

c. Expressivity

As explained in Section 4.1.a, more than one way exists to perform a specific operation such as repeating a set of instructions by using different loop constructs. Despite the fact that it decreases readability as mentioned, it also enhances the expressivity and writability of AYVA++.

4.3 Reliability

Reliability refers to the conformance to specifications, which means whether the language performs to its specifications or not. The evaluation criteria of reliability involves type checking, aliasing, readability and writability.

a. Type Checking

AYVA++ does not have type checking, because we are defining all integers, strings, booleans etc. as an identifier.

b. Aliasing

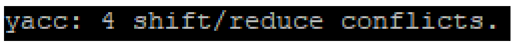
Pointers and references do not exist in AYVA++. Because there is no presence of two or more distinct referencing methods for the same memory location, it provides increased reliability for our language.

c. Readability and Writability

AYVA++ has natural ways to express an algorithm providing a better readability and writability, hence a better reliability as well.

5.0 Handling Conflicts

We encountered four shift/reduce conflicts when Lex and Yacc files are tested together, as below.



```
yacc: 4 shift/reduce conflicts.
```

It was due to a problem related with our recursive boolean expressions. However, we have realized that it was unnecessary, since we can already write Boolean expressions correctly without the need of recursion. After we remove the recursive definition, we saw that all shift/reduce conflicts have disappeared.

6.0 About Precedence and Ambiguity

In our language, comparison signs have a higher precedence than logical operators. To handle with ambiguity, we derived complex statements into simple statements to minimize the rate of mistakes. Finally, we checked our code with example test programs by examining our yacc file and observing the errors generated by parser.

Ultimately, we become sure that the language does not include any ambiguity. For better clarification of the precedence rules, check table 1 below. Operators with higher precedence are given at the top of the table.

( )	Paranthesis
* /	Multiplication / Division
+ -	Addition / Substraction
< <=	Relational less than / less than or equal to
> >=	Relational greater than / greater than or equal to
== !=	Relational is equal to / is not equal to
&&	Logical AND
||	Logical OR
<<	Assignment Operator

Table 1: Precedence of the operators in AYVA++