CS 315 – Programming Languages

PROJECT 1

Cayya Language

Kaan Sancak 21502708 Section-1

Umut Akös 21202015 Section-1

İlhami Kayacan Kaya 21502875 Section-1

Name of the Language: Cayya

The Complete BNF Description of Cayya Language

1. Program

<program> ::= <begin><stmts><end>

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

<stmt> ::= <matched_stmt> | <unmatched_stmt> | <end_stmt>

<end_stmt> ::= <semicolon>

<comment> ::= <hashtag><sentence><endline>

2. Types

<type> ::= <primitiveType>

<primitiveType> ::= “boolean” | “char” | “string” | ‘float’ | “integer” | “digit”

<boolean > ::= <true> | <false> | <dontcare>

<string> ::= <string_identifier><sentence><string_identifier>

<sentence> ::= <word> | <sentence><word> | <sentence><symbol>|<sentence><digit>

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

<char> ::= <char_identifier><letter><char_identifier> |

<char_identifier><digit><char_identifier>

<unsignedInteger> ::= <digit> | <integer><digit>

<signedInteger> ::=<sign><unsignedInteger>

<signedFloat> ::= <signedInteger> .<unsginedInteger> | <signedInteger>.<digit>

<unsignedFloat> :: <unsignedInteger> .<unsginedInteger> | <unsignedInteger>.<digit>

3. Truth Values

<true> ::= “true” | 1

<false> ::= “false” | 0

<dontcare> ::= “Don’t Care” | “X”

4. Constants

<constant> ::= <constantIdentifier><var>

<constantIdentifier> : ~

5. Connectives

<and> ::= &&

<or> ::= ||

<not> ::= ‘ | !

<implies> ::= ->

<equivalence> ::= =>

<iff> ::= ⇔

<equivalenceCheck> ::= ==

<logicalOperation> ::= <and> | <or> | <implies> | <equivalence> | <iff>

<logical_expression> ::= <primary_expression> | < secondary_expression>

<primary_expression> ::= <term> <logicalOperation><term> | <not_expression> |<boolean>

<secondary_expression> ::= <primary_expression> <logicalOperation> <
secondary_expression> | <primary_expression>

<not_expression> ::= <not><term> | <not_expression><term> | <boolean>

6. Variables

<term> ::= <var> | <constant>

<var> ::= <identifier>

<identifier> ::= <word> | <identifier> <digit>|<word><identifier>

7. Predicates

<predicateId> ::= $

<predicateDeclaration> ::=
<predicateId><identifier><LP><parameter_list><RP><LB><predicateBody><RB>

<predicateBody> ::= <return> | <stmts><return>

<return> ::= return<space><assignmentValues>

8. Predicates Corresponding to Function Calls

<predicateInstantiation> ::= run<space><predicate><predicateBody>

<parameter_list> ::= <var_list> | <constant_list> | <var_list><parameter_list> | <constant_list>
<parameter_list>

<constant_list> ::= <constant> | <constant><comma><constant_list>

<var_list> ::= <var> | <var><comma><var_list>

9. Assignment Operator

<assignmentOperator> ::= <equal>

10. Selection

<matched_stmt> ::= if<LP><logical_expression><RP><matched_stmt>else<matched_stmt> |
<while_stmt> | <doWhile_stmt>

<unmatched_stmt> ::= if<LP><logical_expression><RP><stmt> |
if<LP><logical_expression><RP> <matched_stmt> else <unmatched_stmt>

11. Looping

<while_stmt> ::= while<LP> <logical_expression> <RP> <LB> <stmts> <RB>

<for_stmt> ::= for<LP><initialization> ; <logical_expression><RP><LB><stmts><RB>

<doWhile_stmt> ::= do<LB> <stmts> <RB> while <LP> <logical_expression><RP>

12. Input/ Output

<input_stmt>::= cayyin<LP><inputBody><RP>

<inputBody> ::= <boolean> | <char> | <unsignedInt> | <signedInt> | <unsignedFloat> |

<signedFloat> | <string>

<output_stmt> ::=cayyout<LP> <outputBody><RP>

<outputBody> ::= <assignmentValues><plus><outputBody> | <assignmentValues>

13. Declaration and Init

<termDecleration> ::= <type><space><term><endstmt>

<initialization> ::= <term><assignmentOperator><assignmentValues><endstmt>

<termDeclarationwithInit> ::=

<type><space><term><assignmentOperator><assignmentValues><endstmt>

<assignmentValues> ::= <boolean>| <digit> | <char> | <unsignedInt> | <signedInt>

|<unsignedFloat> | <signedFloat> | <constant> |<predicateInstantiation>

14. Symbols

<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'

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

<symbol> :: = <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <comma> | <semicolon> |

<underscore> | <equal> | <dot> | <char_identifier> | <space>

```
<LP> ::= (

<RP> ::= )

<LB> ::= {

<RB> :: = }

<LSB> ::= [

<RSB> ::= ]

<comma> ::= ,

<semicolon> ::= ;

<underscore> ::= _

<equal> ::= =

<string_identifier> ::= “

<char_identifier> ::= ‘

<space> ::= “ ”

<sign> ::= +|-

<hashtag> ::= #

<endline> ::= \n

<dot> ::= .
```

Explanation of Cayya Language Constructions

1. <program> ::= <begin><stmts><end>

This non-terminal state that our program is consisting of statements. This actually is the core of the features that the language is going to present. Our program starts with a "begin" command and ends with the "end" command.

2. <stmts> ::= <stmt> | <stmts> <stmt>

This non-terminal is the representative of the statements that our language consists of. The statements of our language are the lists of the statements.

3. <stmt> ::= <matched_stmt> | <unmatched_stmt>

This non-terminal is created to show the types of the statements that our language consists of. Therefore, the branching according to the statement types occurs after is terminal.

4. <matched_stmt>::=if<LP><logical_expression><RP> <matched_stmt>else<matched_stmt> | <while_stmt> | <doWhile_stmt>

This non-terminal is created to show the syntax of the "if statement" which is matched in terms of the number of the parentheses. A matched-if statement takes another matched-if statement inside and it goes on in the recursive way. By means of this recursive relationship, the number of left parentheses and the right parentheses are kept same and so that the if-statements are matched. Instead of matched-if statement, while-statement can also be included in the body part of the matched-if statement. This does not affect the matching of the if-statements.

5. **<unmatched_stmt>::=if<LP><logical_expression><RP><stmt>if<LP> <logical_expression><RP><matched_stmt>else <unmatched_stmt>**

This non-terminal is created to show the syntax of the “if statement” which is not matched in terms of the number of the parentheses. An unmatched-if statement takes another statement without the concern of whether it is matched-if statement or not. It can also take an if-statement followed by an else-statement which includes unmatched-if statement in its body. It goes on in the recursive way. By means of this recursive relationship, the number of left parentheses and the right parentheses are not concerned.

6. **<while_stmt> ::= while<LP> <logical_expression> <RP> <LB> <stmt> <RB>**

This non-terminal is created in order to show syntax of the “while statement” and this includes logical expression between its two parentheses and this logical expression returns true or false so after and if it returns false while loop continues to work and inside the braces(LB-RB) statements also continues to work until logical expression returns false and if logical expression returns false, while loop finish and stop.

7. **<for_stmt> ::= for<LP><initialization> ; <logical_expression><RP><LB><stmts><RB>**

This non-terminal is created in order to show syntax of the “for statement” and this includes an variable assignment and logical expression between two parentheses returns true or false so after and if it returns false for loop continues to work and inside the braces(LB-RB) statements also continues to work until logical expression returns false and if logical expression returns false, for loop finish and stop.

8. <doWhile_stmt>:: = do<LB> <stmts> <RB> while <LP> <logical_expression><RP>

This non-terminal is created to show the syntax of the “do-while” statement which is a looping statement that first the expressions inside the braces are executed and after the truth value of the logical statement inside the while statement is checked then, according to the result of this check the continuity of the statement execution is determined.

9. <comment> ::= <hashtag><sentence><endline>

This non-terminal is created in order to represent the syntax of the comment statements. By means of this non-terminal, the users of the language will be able to start a comment line by means of putting a hashtag symbol in front of the line. After the comment of the user ends, the program will see a new line in order to stop the comment statement and continue reading the other statements.

10.<type>:: = <primitiveType>

This non-terminal represents different data types of our language. Presently, we have only one sub-data type (primitive type) ;however to make the language more extendable in future, we will use this definition.

11. **<primitiveType> :: = “boolean” | “char” | “string” | ‘float’ | “integer” | “digit”**

This terminal represents different primitive data types in Cayya language. “boolean” represents truth values in three ways, true, false, don’t care. “char” represents character values. “string” represents string. “float” represents double and float values.
“digit” represents digits. ·

12. **<boolean> ::= <true> | <false> | <dontcare>**

This non-terminal defines truth values. “Cayya” language derives Boolean as true, false or dontcare form. True represents terms (explained in other sections) whose truth value is true and true represents terms.

13. **<string> ::= <string_identifier><sentence><string_identifier>**

This non-terminal represents string syntax of “Cayya” language. In Cayya language any string must be in between string identifiers. If Cayya lexer sees any word/sentence between string identifiers (“), it will define these values as strings.

14. **<sentence> ::= <word> | <sentence><word> | <symbol>**

This non-terminal represents sentences (not strings!) which are group of words. This non-terminal will be used to define the difference between identifiers and strings.

15. **<word> ::= <letter> | <word><letter>| <word><digit>**

This non-terminal represents words in “Cayya” language which are group of letters. This non-terminal will be used to define combinations of letters and digits.

16.**<char> ::= <char_identifier><letter><char_identifier> |**

<char_identifier><digit><char_identifier>

This non-terminal represents char values in "Cayya" language. In Cayya language any char character must be in between char identifiers and only one character must be in these identifiers. If "Cayya" lexer sees this notation it will define these representations as char values.

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

This terminal defines digits in "Cayya" language. ·

18.<unsignedInteger> ::= <digit> | <integer><digit>

This non-terminal defines unsigned integers in Cayya language. In Cayya language, unsigned integers can be represented as digit (e.g.. 1, 2,3) or as a collection of digits (e.g.. 123,423).

19. <signedInteger> ::=<sign><unsignedInteger>

This non-terminal represents signed integers in Cayya language, meaning these values are specified as negative or positive. Any signed integer in Cayya language must have a sign symbol followed by an unsigned integer.

20. <signedFloat> ::= <signedInteger> .<unsginedInteger> |

<signedInteger>.<digit>

This non-terminal represents signed float values in Cayya language, meaning that these float values are specified as negative or positive. Any signed float in Cayya language, must have a signed integer followed by unsigned decimal values or a digit.

21.<unsignedFloat> :: <unsignedInteger> .<unsginedInteger> | <unsignedInteger>.<digit>

This non-terminal represents unsigned float values in Cayya language, meaning that these float values are not specified with any sign symbol. Any unsigned float in Cayya language, must have an unsigned integer followed by unsigned decimal values or a digit.

22.<true> ::= "true" | 1

This terminal is used in the process of declaration of the truth value representing "true" for a proposition in the scope of propositional logic.

23.<false> ::= "false" | 0

This terminal is used in the process of declaration of the truth value representing "false" for a proposition in the scope of propositional logic.

24.<dontcare> ::= "Don't Care" | "X"

This terminal is used in the process of declaration of the truth value representing a "Don't care" for a proposition in the scope of propositional logic. This truth value is used to represent the truth value of the proposition which can either be true or false.

25.<constant> ::= <constantIdentifier><var>

This non-terminal is used for constant expression. Cayya language's constants are composed of variable with constant identifier. To our program see the constant, user should use this "~ **"** before the variable, this token is constant identifier and if user use it before the variable like " ~x " (x is a variable) , it will be constant. If constant is taken a value in the program after this, this constant cannot be changed.

26.<constantIdentifier> : ~

This token is a sign to specify constant identifier.

27.<and> ::= &&

This terminal is created to represent the declaration of the "and-operation" that is a connective in the propositional logic which will be used in the combination of two different propositions.

28.<or> ::= ||

This terminal is created to represent the declaration of the "or-operation" that is a connective in the propositional logic which will be used in the combination of two different propositions.

29.<not> ::= ' | !

This terminal is created to represent the declaration of the "not-operation" that is a connective in the propositional logic which will be used in the negation of a particular proposition.

30.<implies> ::= ->

This terminal is created to represent the declaration of the "implies-operation" that is a connective in the propositional logic which will be used in the connection of two different propositions in a way that one propositions truth value is dependent upon the other ones.

31.<equivalence> ::= =>

This terminal is created to represent the declaration of the “equivalence-operation” that is a connective in the propositional logic which will be used in the combination of two different propositions in terms of their similarity. That it, a proposition can be equalized to another one by this connective.

32.<iff> ::= ⇔

This terminal is created to represent the declaration of the “if and only if-operation” that is a connective in the propositional logic which will be used in the combination of two different propositions.

31.<equivalenceCheck> ::= ==

This terminal is created in order to represent the equality but it is different from equal sign “=” because equivalence is written for logical expression in loops like if and while but it works same with implies in terms of checking. After the statement in the loop and returning the logical expression, it checks two terms and provides a Boolean for while loop to return true or false.

32.<logicalOperator> ::= <and> | <or> | <implies> | <equivalence> | <iff>

In Cayya language, user can write a logical operator between two terms. These terms can be constants or variable and operators can be “and”,”or”,”implies”,”equivalence” or “iff”. Therefore, in Cayya language, these are logical operators.

33.<logical_expression> ::= <primary_expression> | < secondary_expression>

In Cayya language, if user use any logical operator between two terms, this would be logical expression and would be primary expression, primary expression is using any logical operator between two terms. However, logical expression can consist of more than two terms so it would be secondary expression like " a && b || c ". To sum up, logical expression can be primary expression or secondary expression by using number of terms.

34.<var> ::= <identifier>

In Cayya language, variable names are defined by their identifiers.

35.<identifier> :: <word> | <identifier> <digit>|<word><identifier>

In Cayya language, identifiers, identifiers are combinations of letters and digits. identifiers cannot start with a digit/integer, but digits/integers can be in the middle or at the end of the identifier. An identifier can be represented by just a one character but cannot be represented by one digit.

SYMBOLS

In Cayya language consists of some symbols. These are letter, LP, RP, LB, RB, LSB, RSB, comma, semicolon, underscore, equal, dot, string identifier, char identifier and space or hashtag. These symbols can be used by writing a program with Cayya language.

36.<predicateId> ::= $

This terminal is created to differentiate the predicate from the other elements of the language, such as constant. This will be considered as the identifier for the predicate.

37.<predicateDeclaration> ::= <predicateId><identifier><LP><parameter_list><RP><LB><predicate Body><RB>

This non-terminal is being used to show how the predicates are declared in our program. It first starts with a dollar sign which is the predicate identifier, then it continues with the identifier to show which predicate it is. After that, it continues with the parameter list enclosed by parentheses. Then, the predicate body is written inside the brackets that are belong to the predicate.

38.<predicateBody> ::= <return> | <stmts><return>

This non-terminal is used to show the body of the predicate. It is seen that the body of the predicate is consisting at least of a return non-terminal. This is used to show the return type of the predicate itself. It can be analyzed that the predicate body can have statements followed by a return statement.

39.<return> ::= return<space><assignmentValues>

This non-terminal is the return of the predicate. This is used to exclude another word for the representation of the return type of the predicate. By means of using that non-terminal, the return type of the predicate will be determined in the body of the predicate. This non-terminal has a “return” reserved word followed by the assignment values.

40. **<predicateInstantiation> ::= run<space><identifier><LP><parameter_list><RP>**

    This non-terminal is created to represent the predicate instantiation and how in the program the predicate is run. To do so, first our language uses a reserved word “run” that will indicate that the following strings will be used in predicate representation. After running, there comes the declaration of the predicate which starts with an identifier, and continues with a parameter list that is enclosed by parentheses.

41. **<parameter_list> ::= <var_list> | <constant_list> | <var_list><parameter_list> | <constant_list> <parameter_list>**

    This non-terminal is created to determine the type of the parameters inside the predicates. The parameter list consists of variable list, and constant list. This non-terminal is used to show the alternatives for the parameter list that will be used in the instantiation of the predicates.

42. **<constant_list> ::= <constant> | <constant><comma><constant_list>**

    This non-terminal is created to construct a list of constants. These will be the list propositions that will be used in the instantiation process of the predicates.

43. **<var_list> ::= <var> | <var><comma><var_list>**

    This non-terminal is created to construct a list of variables. This will be the list that will be used in the instantiation process of the predicates.

44.<termDecleration> ::= <type><space><term><endstmt>

This non- terminal is created to declare the term so that Cayya language can understand what term type is like int, float, string. To create a term, before writing any term, user must write the type of the variable and put the semicolon to end the statement.

45.<initialization> ::=

<term><assignmentOperator><assignmentValues><endstmt>

This non-terminal is created to initialize the term after the declaration. User must write a term firstly and after, assignment operator must be put to providing equality with assignment value and finally semicolon must be put to end the initialization.

46.<assignmentValues> ::= <boolean>| <digit> | <char> | <unsignedInt> |

<signedInt> |<unsignedFloat> | <signedFloat> | <constant>

|<predicateInstantiation>

In Cayya language, boolean, digit, char, usisignedInt, signedInt, unsignedFloat, constant or signedFloat, predicate instantiation are considered as the assignment values. These values will be used in the return process of the predicates. As the predicates may return the predicate instantiation as a value, this may create a recursive relationship if the predicate calls itself. (predicate calling another predicate)

47.<termDeclarationwithInit> ::= <type><space><term><assignmentOperator><assignmentValues><end stmt>

In Cayya language, term declaration and initialization can be written separately but thanks to the term declaration, user can declare and initialize the terms in one line. To be obvious, Cayya can combine initialization and declaration.

48.<input_stmt>::= cayyin<LP><inputBody><RP>

49.<inputBody> ::= <boolean> | <char> | <unsignedInt> | <signedInt> | <unsignedFloat> | <signedFloat> | <string>

In Cayya language, user can enter a input and this input can be between LP and RP. Between the parenthesis, there is input body and input body can be consist of the boolean, char, unsignedInt, signedInt, unsignedFloat, signedFloat or string.

50.<output_stmt> ::=cayyout<LP> <outputBody><RP>

51.<outputBody> ::= <assignmentValues><plus><outputBody> | <assignmentValues>

In Cayya language, program can give a output and this output can be between LP and RP. Between the parantehesis, there is output body and output body can be consist of the boolean, char, unsignedInt, signedInt, unsignedFloat, signedFloat or string.

Non-Trivial Tokens of Cayya Language

- **run :** This token is a reserved word in cayya language. As it can be understood from the name itself it is a reserved word for running the predicates that a declared before. This token cannot be used for identifiers.
- **cayyin :** This token is a reserved word in cayya language. Token can be open as cayya-input(cayyin). Basically it is a reserved word for inputs. Users must use this token before take inputs. This token cannot be used for identifiers.
- **cayyout :** This token is a reserved word in cayya language for giving outputs to user. Token can be open as cayya-output(cayyout). User must use this token before giving outputs. This token cannot be used for identifiers.
- **if:** This token is a reserved word in cayya language for if statements.
- **else:** This token is a reserved word in cayya language for else statements.
- **for:** This token is a reserved word in cayya language for “for statements”.
- **do:** This token is a reserved word in cayya language for “do-while statements”.
- **while:** This token is a reserved word in cayya language for while statements.
- **int:** This token is a reserved word in cayya language for integer type data.
- **float:** This token is a reserved word in cayya language for float type data.
- **string:** This token is a reserved word in cayya language for string type data.
- **char:** This token is a reserved word in cayya language for char type data.
- **return:** This reserved word in cayya language for return statements of predicate calls. Predicates must have a return statement in which predicate returns the given data type.
- **‘~’ sign:** This is a reversed char for constant variable declaration.

Test Programs of Cayya Language

Test Program1:

```
boolean a = true;

boolean b = false;

boolean ~c = false;

boolean result;


result = a && b;


cayyout(result);

result = r | ~c;

cayyout(result);


r = a == b;

r = a =>b;

r = ~c <==> a;

r = a > b;

r = a >= b;

r = a <= b;

r = a ==> b


cayyout( a->b)


cayyin(a)
```

Test Program2:

```
int termVar = 10;


for ( int myDumbVar == 0, termVar > dumbVar )

{

        cayyout("Cayya Language");

        myDumbVar = myDumbVar + 1;

}


float a = 1.0;

float b = -2.2;


int c = -4;

int d = +2;


boolean ac = a < c;

boolean cd = a == d;

boolean taken;


cayyin(taken);


boolean result = taken && ac;


cayyout(result);
```

Test Program3:

```
#This is a string declaration & init

string a = "My Name is cayya";

#This is a char declaration & init

char b = 'c';

int first = 4;

int second;

cayyin(second);

if( first == second){

        cayyout(first);

}

else{

        cayyout(second);

}

dumbPredicate( boolean x, boolean y){

        boolean result = a <==> b;

        return result;

}

boolean f= true;

boolean s = false;

boolean dumb = dontcare;

boolean result = (run dumbPredicate(f,s));

do{

        cayyout(f);

}while( result == false);
```