CS 315 – Programming Languages

PROJECT 2

GROUP ID: 19

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 is start rule of cayya language

<program> ::= <predicateDeclarations><main>

<main> ::= MAIN LP RP LB stmts RB

<predicateDeclarations> ::= <predicateDeclarations><predicateDeclaration>
| <predicateDeclaration>

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

2. Possible Statement Types

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

<matched_stmt> ::= IF LP <logical_expression> RP <matched_stmt> ELSE <matched_stmt>
| <other_stmt>

<unmatched_stmt> ::= IF LP <logical_expression> RP <stmt>
| IF LP <logical_expression> RP <matched_stmt> ELSE <unmatched_stmt>

<other_stmt> ::= <loop> | <declaration> | <init> | <input_stmt> | <output_stmt> | <end_stmt> | COMMENT | <return_stmt>

<end_stmt> ::= ENDSTMT

3. Declarations

<declaration> ::= <varDeclaration> | <constantDeclaration> | <array_declaration>

<varDeclaration> ::= VAR <var_id><end_stmt>

<constantDeclaration> ::=
CONSTANT LP <constantContent> RP <assignmentOp> BOOLEAN <end_stmt>

<constantContent> ::= STRING

4. Predicate

4.1. Predicate Declaration

<predicateDeclaration> ::=
PREDICATE LP <declaration_param_list> RP LB <predicateBody> RB

<declaration_param_list> ::= <declaration_element>
| <declaration_element> COMMA <declaration_param_list>

<declaration_element> ::= VAR <var_id> | CONSTANT

<predicateBody> ::= <return_stmt> | <stmts><return_stmt>

<return_stmt> ::= RETURN <logical_expression><end_stmt>

4.2. Predicate Instantiation

<predicateInstantiation> ::= RUN PREDICATE LP <parameter_list> RP

<parameter_list> ::= <element> | <element> COMMA <parameter_list>

<element> ::= <term> | BOOLEAN

5. Initialization

<init> ::= <varInitialization> | <predicateInstantiation> | <varDecWithInit> | <array_init>
| <array_dec_init> | <assign_element>

6. Array

6.1. Array Declaration

<array_declaration> ::= VAR ARRAY <end_stmt>

6.2 Array Initialization

<array_init> ::= ARRAY ASSIGNMENTOP LB <array_parameter_list> RB <end_stmt>

6.3 Array Declaration and Initialization At The Same Time

<array_dec_init> ::=
VAR ARRAY ASSIGNMENTOP LB <array_parameter_list> RB <end_stmt>

<array_parameter_list> ::= <array_parameter> COMMA <array_parameter_list>
| <array_parameter>

<array_parameter> ::= BOOLEAN | <var_id> | CONSTANT

6.4. Getting Array Element

<array_element> ::= ARRAY LSB <index>RSB

<index> ::= INT | DIGIT

6.5. Assigning Array Element

Logical Expression can be True , False or DontCare

<assign_element> ::= <array_element> ASSIGNMENTOP <logical_expression><end_stmt>

7. Term Initialization

<varInitialization> ::= <var_id> ASSIGNMENTOP <logical_expression><end_stmt>

<varDecWithInit> ::= VAR <var_id> ASSIGNMENTOP <logical_expression><end_stmt>

8. Looping Statements

<loop> ::= <while_stmt> | <for_stmt> | <doWhile_stmt>
<while_stmt> ::= WHILE LP <logical_expression> RP LB <stmts> RB

<for_stmt> ::= FOR LP<varDecWithInit><logical_expression> RP LB<stmts> RB

<doWhile_stmt> ::= DO LB <stmts> RB WHILE LP <logical_expression> RP

9. Logical Expressions(Connectives Relations)

9.1. If and Only If Expression (left associative)

<logical_expression> ::= <logical_expression> IFF <primary_expression>
| <primary_expression>

9.2. Implies Expression (right associative)

<primary_expression> ::= <or_expression> IMPLIES <primary_expression>
| <or_expression>

9.3. Or Expression (left associative)

<or_expression> ::= <or_expression> OR <ternary_expression>
| <ternary_expression>

9.4. Xor Expression (left associative)

<ternary_expression> ::= <ternary_expression> XOR <and_expression>
| <and_expression>

9.5. And Expression (left associative)

<and_expression> ::= <and_expression> AND <not_expression>

| <not_expression>

9.6. Not Expression (unary)

<not_expression> ::= NOT <p_expression>

| <p_expression>

9.7. Paranthesis Expression

<p_expression> ::= LP <logical_expression> RP | <term> | BOOLEAN

| <predicateInstantiation>

10. Term Declaration

<term> ::= <var_id> | CONSTANT

11. Variable Declaration

<var_id> ::= IDENTIFIER

12. Input Statement

<input_stmt> ::= CAYYIN LP <var_id> RP <end_stmt>

13. Output Statement

<output_stmt> ::= CAYYOUT LP <logical_expression> RP <end_stmt>

14. Truth Values

<BOOLEAN > ::= <TRUE> | <FALSE> | <DONTCARE>

<TRUE> ::= true | 1

<FALSE> ::= false | 0

<DONTCARE> ::= dontcare | X

15. Connectives

<AND> ::= &&

<OR> ::= ||

<XOR> ::= ^

<NOT> ::= „ | !

<IMPLIES> ::= ->

<IFF> ::= < == >

Explanation of Cayya Language Constructions

Program Structure

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

To have a valid cayya code, there are two major components. The first one is predicate declarations in which programmer should define its methods by using the specific syntax described in predicate declaration part. The second component is the main in which program starts to execute.

- **<predicateDeclarations> ::= <predicateDeclarations><predicateDeclaration>| <predicateDeclaration>**

If the programmer wants to declare any predicate, she/he must declare it before the main

function. The predicate description syntax defined below.

- **<main> ::= MAIN LP RP LB stmts RB**

main is one of the most important non-terminals of cayya language, it the scope in which code executes. Except of the predicate declarations every piece of code that makes assignments or declarations must go in to main's scope.

Possible Statement Types

- **<stmts> ::= <stmt> | <stmts> <stmt>**

This non-terminal is created to show the types of the statements that our language consists of. Our language has matched, unmatched, end, comment and return types of statements. Therefore, the branching according to the statement types occurs after is terminal.

- **<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.

- **<matched_stmt> ::= IF LP <logical_expression> RP <matched_stmt> ELSE <matched_stmt> | <other_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.

- **<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.

- **<other_stmt> ::= <loop> | <declaration> | <init> | <input_stmt> | <output_stmt> | <end_stmt> | COMMENT | <return_stmt>**

This non-terminal represents every other statement that can be done in our language including declarations of variables and constants, input output statements, looping statements…

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

  This is the terminating statement of our language.

Looping Statements

- **<loop> ::= <while_stmt> | <for_stmt> | <doWhile_stmt>**

There are there looping statements in our language which are while, for and do while. Detailed explanations are given below

- **<while_stmt> ::= WHILE LP <logical_expression> RP LB <stmts> 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.

- **<for_stmt> ::= FOR LP <varDecWithInit><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.

- **<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.

Truth Values

- **<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. We did not explained truth values of true, false and don’t care for briefty.

Connectives

- **<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.

- **<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.

- **<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.

- <**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.

- **<XOR>**::= ^

This terminal is created to represent the declaration of the “XOR” that is a connective in the propositional logic which will be used in the xor of two different propositions in terms of their similarity and differentiate.

- **<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.

Logical Expressions

- **If and Only If Expression (left associative)**

<logical_expression> ::= <logical_expression> IFF <primary_expression>

| <primary_expression>

This non-terminal is created to show the syntax of the “if and only if” expression. As it has the lowest precedence the logical expressions start with the iff expression. As it continues by extending the logical expression to the other expressions in the ascending order of the precedence, the logical expression becomes more complex expressions.

- **Implies Expression (right associative)**

<primary_expression> ::= <or_expression> IMPLIES <primary_expression>

| <or_expression>

This non-terminal is created to show the syntax of the “implies” expression. As it has the second lowest precedence, the logical expressions continues with the implies expression after iff expression. As it continues by extending the logical expression to the other expressions in the ascending order of the precedence, the logical expression becomes more complex expressions.

- **Or Expression (left associative)**

<or_expression> ::= <or_expression> OR <ternary_expression>

| <ternary_expression>

This non-terminal is created to show the syntax of the “or” expression. As it has the thirdlowest precedence, the logical expressions continues with the or expression after implies expression. As it continues by extending the logical expression to the other expressions in the ascending order of the precedence, the logical expression becomes more complex expressions.

- **Xor Expression (left associative)**

<ternary_expression> ::= <ternary_expression> XOR <and_expression>

| <and_expression>

This non-terminal is created to show the syntax of the “xor” expression. As it has the forth lowest precedence, the logical expressions continues with the xor expression after or expression. As it continues by extending the logical expression to the other expressions in the ascending order of the precedence, the logical expression becomes more complex expressions.

- **And Expression (left associative)**

<and_expression> ::= <and_expression> AND <not_expression> | <not_expression>

This non-terminal is created to show the syntax of the “and” expression. As it has the second highest precedence, the logical expressions continues with the implies expression after xor expression. As it continues by extending the logical expression to the other expressions in the ascending order of the precedence, the logical expression becomes more complex expressions.

- **Not Expression (unary)**
- <not_expression> ::= NOT <p_expression> | <p_expression>

This non-terminal is created to show the negation expression. This means that we have an expression and this expression is a boolean type and if we write NOT in front of the expression, this expression would be opposite of its old value. If it is true and we write NOT, it would be false or vice versa.

- **Paranthesis Expression**
- **<p_expression> ::= LP <logical_expression> RP | <term> | BOOLEAN**
  | <predicateInstantiation>

This non-terminal is created to show the syntax of the “paranthesis” that will enclose the expressions. As the paranthesis has the highest value in the precedence order, it is declared at very end in the logical expressions branch. Paranthesis expression can be variable, constant, or boolean. By means of this assignments to the paranthesis expression, the return values of the logical expressions (TRUE or FALSE) can be reached as the code goes at the very bottom from the logical expression’s branch. Logical expressions can also be assigned to the term values which will lead to an opportunity to include terms inside the looping statements, matched and unmatched statements, return statement, and array element assignments.

Declarations

There are three types of declarations in cayya language which can be done in main function(other than predicate declaration which must be done outside the main)

- **<declaration> ::= <varDeclaration> | <constantDeclaration> | <array_declaration>**

These three declaration types are variable(var) declaration constant declaration and array declaration(detailed explainaion in array description).

INITIALIZATIONS

<init> ::= <varInitialization> | <predicateInstantiation> | <varDecWithInit> | <array_init>
| <array_dec_init> | <assign_element>

There are different types of initializations in cayya language which are array initialization, variable initialization, array element assignment. Also this non terminal encapsulates some non-terminals like var declarations with initializations and array declarations with initializations and also predicate runs(executing a predicate).

Variables

In cayya language variable represent variables whose truth values can be assigned multiple times.A variable declaration has 2 main components which are keyword VAR(“var”) and a identifier for the variable name. In Cayya language, variable names are defined by their identifiers

- **<var_id> ::= IDENTIFIER**

Variable Declarations

- **<varDeclaration> ::= VAR <var_id><end_stmt>**

In cayya language variable represent variables whose truth values can be assigned multiple times. For example:

var flag;

is a simple way to define a variable.

Variable Initializations

There are two ways to initialize a variable. Init can be done during declaration and can be done after declaration. A variable's value can be assigned to any logical_expression(connective expressions or boolean or a constant's value) multiple times during the execution.

- **<varInitialization> ::= <var_id> ASSIGNMENTOP <logical_expression><end_stmt>**

An already defined var can be assigned to a value by using assignment operator.

For example:

flag = true;

flag = true && false || (false ^ false);

is some simple ways to init a var.

- **<varDecWithInit> ::= VAR <var_id> ASSIGNMENTOP <logical_expression><end_stmt>**

A var can be assigned to a truth value during declaration.

For example:

var flag = a -> b; (assuming a and b declared and init before)

Constants

A constant rule in our lex file has the structure of:

<CONSTANT> ::= <CONSTANTIDENTIFIER><IDENTIFIER> where

<CONSTANTIDENTIFIER> ::= '~'

How to Declare a Constant

- **<constantDeclaration> ::=CONSTANT LP <constantContent> RP <assignmentOp> BOOLEAN <end_stmt>**

In cayya language constant variables are the variables whose truth values can be assigned ones and remains as the same until the end of the program execution. That's why a constant variable's truth value can only assigned ones in cayya language which must be done during declaration of the variable. A constant variable has a string content which again must be declared during declaration of the constant. Also there is no strict key word for constant(like in the variables there

is ‘var’) but there is a key character which must be put at the beginning of constant identifier(which is ‘~’)

For example:

~man(“Man is Mortal”);

is a simple way to declare a constant

- **<constantContent> ::= STRING**

This non-terminal represents that a constant content must be a string

ARRAYS

How to Declare an Array

- **<array_declaration> ::= VAR ARRAY <end_stmt>**

In cayya language there is no constant array which means every array element can be changed during execution that’s why there is a keyword ‘var’ at the beginning of array declaration which must be followerd by a array keyword(@) and identifier.

For example:

var @arr;

is a simple way to declare an array.

How to Init an Array

As in the case of var there are two ways to init an array in cayya language. A init can be done for a defined array or can be done during a declaration of array

- **<array_init> ::= ARRAY ASSIGNMENTOP LB <array_parameter_list> RB <end_stmt>**

This non-terminal represent array initialization for already defined array. Basically array takes a list of truth values.

For example:

@arr = { true, false, false, true}; is a simple example.

- **<array_dec_init> ::= VAR ARRAY ASSIGNMENTOP LB <array_parameter_list> RB <end_stmt>**

This non-terminal represent array initialization during a declaration of an array.

For example:

var @arr = { true, false, false, true}; is a simple example.

- **<array_parameter_list> ::= <array_parameter> COMMA <array_parameter_list> | <array_parameter>**

This non terminal represent array parameter list which is the list an array index could take

- **<array_parameter>** ::= BOOLEAN | <var_id> | CONSTANT

This non terminal represent possible values for an array parameter which can be a var, truth val or a constant. When array index assigned to a var or constant the index takes the value of constant or var as its own truth value.

Array Elements

How to Reach Array Elements

- **<array_element> ::= ARRAY LSB <index>RSB**

An array element can be reached by writing the index of the element between '[' ']' symbols after array name. For example:

@arr[4] is the correct way to reached the 4^{th} index of the array

- **<index> ::= INT | DIGIT**

An array index can be an integer(0 and positive integers)

How to Change Array Element's

- **<assign_element> ::= <array_element> ASSIGNMENTOP <logical_expression><end_stmt>**

An array index can be assigned any truth value by reaching the element and using assignment operator.

For example:

@arr[4] = dontcare; assigns the 4th index of the array as dontcare truth value.

PREDICATES

PREDICATE DECLARATION

- **<predicateDeclaration> ::= PREDICATE LP <declaration_param_list> RP LB <predicateBody> 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. This is shown with the "predicate" token. After that, it continues with the parameter list enclosed by parentheses. Then, the predicate body is written inside the brackets that are belongs to the predicate.

For example:

```
$foo( var x, var y){

return x ^ y;

}
```

- **<declaration_param_list> ::= <declaration_element>**

  | <declaration_element> COMMA <declaration_param_list>

This non-terminal is being used to show the list of the parameters that are used in the declaration

process of the predicates. The list consists of either one declaration element or one declaration element followed by a list of declaration elements. This list is separated by the comma in the process of declaration enabling the parameter list to declare more that one parameter.

- **<declaration_element> ::= VAR <var_id> | CONSTANT**

This non-terminal is being used to show the types of the declaration element that is to show whether it is a variable specified by the “var” token or it is constant specified by the “constant” token.

- **<predicateBody> ::= <return_stmt> | <stmts><return_stmt>**

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.

- **<return_stmt> ::= RETURN <logical_expression><end_stmt>**

This non-terminal is used to show how the statements are returned. The syntax of the return type is starting with the “return” token which simply is the reserved word for return, and continues by the logical expressions. These are sequenced in a way in terms of the precedence of the logical expressions (connectives) at the end of all the expressions it is structured in a way that it end up with **boolean** values.

How to Instantiate a Predicate

The predicates can be run as follows:

- **<predicateInstantiation> ::= RUN PREDICATE LP <parameter_list> RP**

This non-terminal is created to show the syntax of the predicate statement. The instantiation of the predicate starts with the reserved word “run” and then it continues with the predicate which consists of the predicate id and the identifier that identifies the predicate. After these, a parameter list is defined inside the parenthesis.

For example:

run $foo(true, ~y); runs the predicate with parameters true and constant y

- **<parameter_list> ::= <element> | <element> COMMA <parameter_list>**

This non-terminal is created to show how the parameter list is declared. Parameter list consists of one or more “element”s. Those elements are in the form of whether term (which is either identifier or constant) or the boolean values. Here the predicate gets the TRUE or FALSE values.

- **<element> ::= <term> | BOOLEAN**

This non-terminal is used to show the types of the elements are in the form of whether term (which is either identifier or constant) or the boolean values.

- **<term> ::= <var_id> | CONSTANT**

In Cayya language, term declaration can be written separately but thanks to the term declaration, user can declare the terms in one line and this declaration can be composed of variable identifier and constant.

Input Output Statements

- **<input_stmt> ::= CAYYIN LP <var_id> RP <end_stmt>**

If input statement is created, it should be composed of CAYYIN left paranthesis and variable identifier and right paranthesis and end_stmt.

- **<output_stmt> ::= CAYYOUT LP <logical_expression> RP <end_stmt>**

In Cayya language, program can give a output and this output can be between LP and RP. Between the parenthesis, there is output body and output body can be consist of the logical expression and these logical expression possibilities can be seen logical expression part

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”.
- **var:** This token is a reserved word in cayya language for variables
- **while:** This token is a reserved word in cayya language for while statements.
- **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.
- **main:** main is a reserved word for Main function.
- **“~“ sign:** This is a reversed char for constant variable declaration.
- **“$“ sign:** This is a reversed char for predicates.
- **“@“ sign:** This is a reversed char for arrays.

How Presedence and Ambiguity Handled

The precedence rules of the logical expressions (connectives) are implemented by means of the orientation of these terminals (and, or, implies, etc.) in the yacc and BNF structure of the language. The precedence order (increasing order) of the connectives and the parenthesis are as follows: iff, implies, or, xor, and, parentheses. As they are declared one after another in the yacc file, when the yacc file is traversed the program will first meet iff structure then implies structure and then or structure etc, until the parentheses. As it comes to very end and traverse the parentheses structure, it either gets the values for the types like variable, constant, boolean or it gets the parentheses and recurse inside the logical operations ending up with creating a logical formula with connectives. In terms of solving the ambiguities, apart from ordering of the expressions which are complex as the branch of the expressions is going into so deep, we checked all of the order in the yacc file in order to get no ambiguities in our language. In the compilation of yacc file, we used some flags in order to declare the places of the conflicts. By means of detecting them, we resolved our issues about the ambiguities and conflicts that we have confronted along the way of creating our language.