Bilkent University

Department of Computer Engineering

CS315 – Programming Languages
Project #2 Report

Group #49

Group Members:

Pelin Elbin Gunay – 21402149 – Section #3

Hareem Larik – 21503645 – Section #3

Emine Ayse Sunar – 21502099 – Section #3

Date: Nov 20, 2017

Contents

Language Design

Language Name

PERSEID

BNF Description

<program> → class <class_identifier> { <block> **[main]** }

<class_identifier> → <upper_case_letter><string_char>

<upper_case_letter> → [A-Z]

<string_char> → <char>

| <char> <string_char>

<char> → <letter> | <digit>

<space> →

<letter> → <upper_case_letter> | <lower_case_letter>

<lower_case_letter> → [a-z]

<digit> → [0-9]

<block> → <assign_const> <block>

| <const_declaration> <block>

| **<assign_predicate>** <block>

| <statement> <block>

|

<statements> → <statement>

| <statement> <statements>

<statement> → <assign_var>

| <re_assign_var>

| <conditional_statement>

| **<predicate_call>**

| <loop_statement>

| <input_statement>

| <output_statement>

| <comment_statement>

| <var_declaration>

| <array_assignment>

| <array_declaration>

<comment_statement> → <single_line_comment>

<single_line_comment> → // <const_char>

<assign_const> → const <constant_identifier> <assignment_op> <compound_proposition> ;

<constant_identifier> → " <const_char> "

<const_char> → <string_char> <const_char>

|

<assignment_operator> → =

<var_declaration> → var <var_identifier> ;

<const_declaration>→ const <constant_identifier> ;

<compound_proposition>→ <truth_value> <connective> <truth_value>

| <proposition> <connective> <proposition>

| <proposition> <connective> <truth_value>

| <truth_value> <connective> <proposition>

| <proposition>

| <truth_value>

| <negation> <proposition>

| <negation> <truth_value>

| (<compound_proposition>) ;

<connective> → <and_connective>

| <or_connective>

| <implies_connective>

| <ifandonlyif_connective>

<and_connective> → &

<or_connective> → |

<implies_connective> → ->

<ifandonlyif_connective> → <->

<negation> → !

<proposition> → <constant_identifier>

| <var_idenfier>

<truth_value> → true | false

<assign_predicate> → predicate truthvalue <predicate_identifier> (<parameters>){ <statements> <return_statement> }

<predicate_identifier> → <string_char>

<parameters> → <parameter>

| <parameter> , <parameters>

|

<parameter> → <compound_proposition>

| *var* <var_identifier>

<assign_var> → var <var_identifier> <assignment_op> <compound_proposition> ;

| var <var_identifier> <assignment_op> <input_statement> ;

<var_identifier> → <string_char>

<re_assign_var> → <var_identifier> <assignment_op> <compound_proposition> ;

| <var_identifier> <assignment_op> <input_statement> ;

<conditional_statement> → <if_statement>

<if_statement> → if (<logic_expr>) { <statements> }

| if (<logic_expr>) { <statements> } else { <statements> }

<logic_expr> → <compound_proposition> <comparison>

<compound_proposition> | <compound_proposition>

<comparison> → == | !=

<array_declaration> → array truthvalue <array_identifier> [ ] ;

<array_identifier> → <string_char>

<predicate_call> → <predicate_identifier> (<parameters>) ;

<loop_statement> → while (<logic_expr>) { <statements> }

<return_statement> → return <**truth_value**> ;

<input_statement> → input:

<output_statement> → output: (<output_strings>);

<output_string> → <string_char> + <output_string>

| <string_char>

<main> → main ( ) { <statements> }

<array_assignment> → array truthvalue <array_identifier> [ ]
<assignment_operator> [<truth_values>] ;

<truth_values> → <truth_value>

| <truth_value> , <truth_values>

Note: The differences from the previous report are written in bold (predicate definitions, predicate declarations, variable declarations, constant declarations, array declaration, array assignment, main)

Non-Terminals specified in Project #1

<program> = is the start symbol of our programming language, it represents a complete program.*<program>* always defines a class.

<class_identifier> = is the name given to a class. It will always start with an upper case letter.*class* is a reserved word.

<upper_case_letter>= representsany of the letters of the English alphabet in upper case.

<lower_case_letter>= defines any of the letters of the English alphabet in lower case.

<string_char>= defines a series of characters or words. It can either be a character or a word with numbers or letters.

<char>= represents a character, it can be a digit or a letter.

<space>= represents space.

<letter>= represents a letter of the English alphabet, it could be an upper case letter or a lower case letter.

<digit>= represents any digit from 0-9.

<block> = represents a block of code inside a class. It can either be a block of code (for recursion), a declaration of a constant type, the code of a function declaration or it can be a series of statements.

<constant_assignment>= represents the declaration of a constant type. To make it recursive, it can either be a single declaration or a series of declaration of constants.

<function_assignment>= represents the code of a function. To make it recursive, it can either be a single function or a series of function declarations, one after another.

<statements>= represents a series of 'statement's. To make it recursive, it can either be a single statement or a series of statements, one after another.

<statement>= represents a single statement. It can be the declaration of a variable of any type, the reassignment of a value to an already declared variable, a conditional statement, a function call, a loop declaration oran statement that asks for input or gives output.

<comment_statement>= represents a comment. It could be empty, a single line comment or a multiple line comment.

<single_line_comment>= represents a comment that is written on one line.

<multiline_comment>= represents a comment that is written on more than one line.

<assign_const>= represents the assignment of a constant.*const* is a reserved word. It can be assigned any value which will be immutable.

<constant_identifier>= represents the name given to a constant while its declaration. It will be a string.

<const_char>= represents a string name given to a constant. It could be a string or a

<assignment_operator>= represents the operator "=". It is used to assign values to variable or constant types.

<compound_proposition>= represents a constant/variable type or the result of operations between two variable/constant types. It can be the result of a operation between two compound propositions, the negation of a compound proposition, a boolean value, or compound proposition (for recursion).

<connective>= represents a operation done on two variables/constants. It can be an AND(&), OR(|), implies(->) or if and only if(<->) operation.

<and_connective>= represents the AND operation.

<or_connective>= represents the OR operation.

<implies_connective>= represents the IMPLIES operation.

<ifandonlyif_connective>=represents the IFANDONLYIF operation.

<negation>= represents the negation operation, which negates the value of any variable.

<proposition>= represents a variable or a constant.

<truth_value>= represents the boolean values of true and false

<assign_function>= represents the declaration of a function. *func* is a reserved word. It has a return type, a name, parameters in parenthesis, and function statements and return statement in curly brackets.

<return_type>= represents the return type of a function. it can be void(nothing) or a truth value.

<function_identifier>= represents the name given to a function while its declaration. It will be a string.

<parameters>= represents a series of parameters in the prototype of a function. It can be a single parameter or a series of parameters for recursion.

<parameter>= represents a single parameter in the prototype of a function. It can be a variable or a constant.

<var_assignment>= represents the assignment of a variable. It can be a single variable assignment or multiple variable assignments.

<assign_var> = represents the declaration of a variable. *var*is a reserved word. It can be assigned a compound proposition.

<var_identifier>= represents the name given to a variable while its declaration. It will be a string.

<re_assign_var> = represents the reassignment of an already declared variable.

<conditional_statement>= represents a conditional if-statement.

<if_statement>= represents a conditional if statement. It is either going to be an if condition or an if-else condition.

<logic_expr>= represents the comparison between two compound propositions.

<comparison>= represents the logic comparison of “==” or ”!=”.

<function_call>= represents a call to a function, it is the name of the function and parameters in parenthesis.

<loop_statement>= represents a loop declaration. It can only be a while loop.

<while_loop> = represents a while loop. *while*is a reserved word. The statements in the braces will keep on executing until the logic expression turns false.

<empty> = represents empty space.

<return_statement>= represents the return statement at the end of each function. it can be a compound proposition or void.

<input_statement>= represents the input taken from the user.

<output_statement>= represents the output given to console. It will be a string.

<output_string>= represents the string output given to the console. For recursion, it can be string or compound proposition.

New Non-Terminals Specified in Project #2

<main>= represents the start of the execution

<var_declaration>= represents the declaration of a variable. (e.g. var var1;)

<const_declaration>= represents the declaration of a constant. (e.g. const "const1";)

<array_declaration>= represents the declaration of an array. (e.g. array truthvalue temp[];)

<predicate_calls>= represents a predicate call such as predicateName(); where predicateName is the name of the predicate function.

<assign_predicate>= this represents a predicate function which only returns truth values, in other words Booleans –true, false.

<array_assignment>= represents the assignment of an array. (e.g. array truthvalue arrayName[] = [true, false];)

Conventions:

- Reserved words: *while, predicate, true, false, if, else, return, const, var, truthvalue, input, output, class*, ***main, array.***
- The comments will only be single line comments. They will begin with "//" symbol.
- The constants are only going to be assigned once. They have to be assigned a specific value meaning they cannot be empty. When assigning a constant, the declaration must start with a reserved word *const* followed by the constant identifier. The constant identifier is going to be between two "" (quotation marks), e.g "manIsMortal". **In addition, they cannot be declared inside a predicate and inside the main function. Also while defining constant there shouldn't be any spaces between the quotation marks. (i.e "man is mortal" will be a syntax error, instead "manIsMortal" could be used.)**
- In the lexical analysis, we can use all the ASCII characters in the constants, but we did not show that in our BNF to shorten the BNF description, otherwise we had to add 128 lines for each ASCII character.
- The variables could be assigned multiple times. While assigning it for the first time, the declaration should start with the reserved word *var*. The variable identifier cannot contain spaces.
- We only have truth values as literals, i.e, *truthvalue*, it can be *true* or *false*. Since this programming language is only used for propositional calculus, it is more efficient to only have literals of type *truthvalue.*

- The identifier for class will always begin with an upper case letter. The identifier for variable will not have spaces in between, and can contain uppercase, lowercase letters and digits. The identifier for predicate functions will have the same properties as the variable identifiers. In the lexical analysis, the function and the variable identifier will both be identified as the same kind of identifier.
- **The variable names and constant names could not be reserved words such as const, var, main etc. Using reserved words at variable or constant declarations will cause a syntax error.**
- **In the predicate and main functions, there could be no predicate function declarations.**

Note: The changes made in the second project are written in bold.

Non-Trivial Tokens Used in the Grammar

- **while:** the token for the start of the while loop.
- **predicate:** the token for the declaration and assignment of predicate functions
- **true:** the token for the truthvalue (boolean) true.
- **false:** the token for the truthvalue (boolean) false.
- **if:** the token used at the start of the if statement.
- **else:** the token used at the else statement.
- **return:** the token used at the start of the return statement.
- **const:** the token used at the constant declarations and assignments.
- **var:** the token used at the variable assignments and declarations.
- **truthvalue:** the token representing Boolean values.
- **input:** the token used when getting inputs.
- **output:** the token used when getting outputs.
- **class:** the token used at the beginning of a class.
- **main:** the token used at the beginning of the main function.
- **array:** the token used at the array assignments and array.

Readability/Writability/Reliability:

Readability: Our programming language has been designed in such a way that it uses the conventional reserved words and the syntax of popular programming languages, this will increase the readability since the programmer will already be familiar with these keywords and syntax from other programming languages.

Writability: This programming language was created only for the purpose of propositional calculus, thus it only serves the specific needs of propositional calculus. For example, there are not any unnecessary literals such as integers, since intergers are not used in propositional calculus.

Reliability: We tried to design our programming language in such a way that there will be the least amount of errors.

How We Resolved the Conflicts?

Our grammar does not have any conflicts, however at the beginning we had 21 shift/reduce conflicts. Here we explain how we resolved these conflicts.

- **From 21 shift/reduce conflicts to 13 shift/reduce conflicts:** Initially we had 21 shift/reduce conflicts. We figured out that shift/reduce conflicts are usually caused because of recursions. Thus we made some changes in our <compound_proposition> non-terminal to reduce the recursion. This was our initial compound_proposition in yacc:
  ➔ compound_proposition: compound_proposition connective compound_proposition | negation compound_proposiiton | truth_value | propoisiton

  As it could be seen, there are 3 recursions. To avoid this, we changed the rule to this:

  ➔ compound_proposition: proposition connective proposition | truth_value connective truth_value | proposition connective truth_value | truth_value connective proposition | negation proposition | negation truth_value | truth_value | propoisiton

  As it could be seen, we have reduced the number of recursions. After this change, our shift/reduce conflicts have reduced to 13.

- **From 13 shift/reduce conflicts to 0 shift/reduce conflicts:** Later on, we recognized that we were making double recursions, which were eventually causing shift/reduce conflicts. We were having double recursions in our *block* and *statements* rules in yacc. In order to resolve this we canceled one of the recursions. Initially we had these two rules:
  ➔ block: statements block | assign_constant block | assign_predicate block |
  ➔ statements: statement | statement statements

  As it could be seen, there is recursion in both the block and statements rules. Thus the parser could not decide which recursion to trace, thus it causes a shift/reduce conflict. To resolve this, we changed the block rule so that it would call a single statement, instead of calling the statements rule. We changed the lock rule to this:

  ➔ block: statement block | assign_constant block | assign_predicate block |

  So, now when the parser wants to add multiple statements, it would only use the recursion in the block rule. After making this change we have reduced our conflicts from 13 to 0.

Important Note

We do not have a main function in our lexer file because we already have a main function in our yacc file. Since we are including the lex.yy.c file in our yacc file, it was causing an error saying that there are two main functions, thus we removed the main function in our lexer file. However our lexer file compiles when a main function is added.