Bilkent University

Department of Computer Engineering

CS315 – Programming Languages

Project #1 Report

Group #49

Group Members:

Emine Ayse Sunar – 21502099 – Section #3

Hareem Larik – 21503645 – Section #3

Pelin Elbin Gunay – 21402149 – Section #3

Date: Oct 23, 2017

Contents

[Part A: Language Design 3](#_Toc496558381)

[Language Name 3](#_Toc496558382)

[BNF Description 3](#_Toc496558383)

[Non-Terminals 5](#_Toc496558384)

[Conventions: 7](#_Toc496558385)

[Readability/Writability/Reliability: 8](#_Toc496558386)

[Part B: Lexical Analysis 8](#_Toc496558387)

[Part C: Example Programs 11](#_Toc496558388)

# Part A: Language Design

## Language Name

PERSEID

## BNF Description

<program>🡪 class <class\_identifier> { <block> **[main]** }

<class\_identifier>🡪 <upper\_case\_letter><string\_chars>

<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\_calls**>

| <loop\_statement>

| <input\_statement>

| <output\_statement>

**| <var\_declaration>**

**| <const\_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> }**

<function\_identifier>🡪 <string\_chars>

<parameters> 🡪<parameter>

|<parameter> , <parameters>

|

<parameter> 🡪<compound\_proposition>

|*var* <var\_identifier>

<assign\_var> 🡪 var <var\_identifier><assignment\_op><compound\_proposition> ;

<var\_identifier>🡪<string chars>

<re\_assign\_var>🡪 <var\_identifier> <assignment\_op> <compound\_proposition>;

<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 >

**<array\_declaration> 🡪array truthvalue <array\_identifier>[ ] ;**

<comparison> 🡪 == | !=

**<predicate\_call>🡪 <predicate\_identifier> (<parameters>);**

<loop\_statement>🡪<while\_loop>

<while\_loop> 🡪while ( <logic\_expr> ){ statements }

<return\_statement>🡪return <**truth\_value**>;

<input\_statement>🡪input:

<output\_statement>🡪output: ( <output\_strings> );

<output\_string>🡪<string\_chars>+<output\_strings>| <string\_chars><compound\_proposition>+ <output\_strings><compound\_proposition>

**<main>🡪 main() {<statements>}**

**<array\_assignment>🡪array truthvalue <array\_identifier> [ ] <assignment\_operator> [truth\_value{,<truth\_value>}];**

**<array\_identifier>🡪 <string\_chars>**

## Non-Terminals

**<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.

## Conventions:

* + - * Reserved words: *while, func, true, false, if, else, return, const, var, void, truthvalue, input, output, class.*
      * The comments will only be single line comments. They will begin with “//” symbol.
* The constants are only going to be declared once. They have to be assigned a specific value meaning they cannot be empty. When assigning a constant, the declaration must start with a a reserved word *const* followed by the constant identifier. The constant identifier is going to be between two “”(quotation marks), e.g “man is mortal”. In addition, they cannot be declared inside a function.
* 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 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 identifier for constants may have spaces in between, furthermore, constant identifiers will be between “”(quotation marks)and can contain uppercase, lowercase letters and digits.

### 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.

# Part B: Lexical Analysis

SPACE [ \t]

NEWLINE [\n]

SINGLELINECOMMENT \/[/]+.\*

DIGIT [0-9]

LOWERCASE [a-z]

UPPERCASE [A-Z]

LEFTPARANTHESIS \(

RIGHTPARANTHESIS \)

LEFTCURLY \{

RIGHTCURLY \}

LEFTSQUARE \[

RIGHTSQUARE \]

DOT \.

SEMICOLON \;

COLON \:

COMMA \,

ASSIGN =

EQUALS ==

NOTEQUAL !=

AND \&

OR \|

IMPLIES \->

IFANDONLYIF <\->

NEGATION !

CONNECTIVE [{AND}{OR}{IMPLIES}{IFANDONLYIF}]

CONCATENATION \+

IF if

ELSE else

TRUE true

FALSE false

TRUTH\_VALUE {TRUE}|{FALSE}

WHILE while

STRING \"([^\\\"]|\\\"|\\\n|\\\\)\*\"

ALPHANUMERIC {LOWERCASE}|{UPPERCASE}|{DIGIT}

CONSTANT\_IDENTIFIER \"[^\"\n]\*\"

CLASS\_IDENTIFIER {UPPERCASE}{ALPHANUMERIC}\*

FUNCTION func

CONSTANT const

VOID void

TRUTHVALUE truthvalue

RETURN\_TYPE {VOID}|{TRUTHVALUE}

VARIABLE var

INPUT input{COLON}

OUTPUT output{COLON}

CLASS class

RETURN return

IDENTIFIER {LOWERCASE}{ALPHANUMERIC}\*

%%

{SEMICOLON} printf("<SEMICOLON>");

{COMMA} printf("<COMMA>");

{NEWLINE} printf("\n");

{CONSTANT\_IDENTIFIER} printf("<CONSTANT\_IDENTIFIER>");

{WHILE} printf("<WHILE>");

{IF} printf("<IF>");

{ELSE} printf("<ELSE>");

{EQUALS} printf("<EQUALS>");

{NOTEQUAL} printf("<NOTEQUAL>");

{CLASS} printf("<CLASS>");

{RETURN} printf("<RETURN>");

{LEFTPARANTHESIS} printf("<LEFTPARANTHESIS>");

{RIGHTPARANTHESIS} printf("<RIGHTPARANTHESIS>");

{LEFTCURLY} printf("<LEFTCURLY>");

{RIGHTCURLY} printf("<RIGHTCURLY>");

{INPUT} printf("<INPUT>");

{OUTPUT} printf("<OUTPUT>");

{VARIABLE} printf("<VARIABLE>");

{FUNCTION} printf("<FUNCTION>");

{CONSTANT} printf("<CONSTANT>");

{SINGLELINECOMMENT} printf("<SINGLELINECOMMENT>");

{CLASS\_IDENTIFIER} printf("<CLASS\_IDENTIFIER>");

{SPACE} printf(" ");

{ASSIGN} printf("<ASSIGN>");

{AND} printf("<AND>");

{OR} printf("<OR>");

{IMPLIES} printf("<IMPLIES>");

{NEGATION} printf("<NEGATION>");

{IFANDONLYIF} printf("<IFANDONLYIF>");

{TRUE} printf("<TRUE>");

{FALSE} printf("<FALSE>");

{RETURN\_TYPE} printf("<RETURN\_TYPE>");

{IDENTIFIER} printf("<IDENTIFIER>");

%%

int yywrap(){

return 1;

}

int main(void){

yylex();

return 0;

}

# Part C: Example Programs

**Test1**

class Test1{

//Testing cont and var instantiation

const "const 1" = true;

const "const2" = false;

var var1 = "const1" & "const2";

}

**Test2**

class Test2{

//Testing function declarations and conditional statements

const "const1" = true;

const "const2" = false;

var var1 = "const1" | "const2";

var var2 = "const2" & var1;

func truthvalue funcname(var1, var2){

if(("const1" & "const2") == var1){

var2 = true;

}

else{

var2 = false;

}

return var2;

}

}

**Test3**

class Test3{

//testing function calls

var var1 = true;

var var2 = false;

func void funcname(var parameter){

parameter = false;

}

funcname(var1);

}

**Test4**

class Test4{

//testing connectives

const "const1" = true;

var p = false;

var a = "const1" & p;

var b = "const1" | p;

var c = "const1" -> p;

var d = "const1" <-> p;

var e = !"const1";

}

**Test5**

class Test5{

//testing variable reassignment and while loop and input-output

var p = true;

p = false;

while(p != true){

p = true;

}

p = input:

output:(p);

}