PACK007 PART 2 Report

THE ELEGANT WAY OF CODING

MEHMET TAHA CETIN	21400281	SEC:01
SEDA GULKESEN	21302403	SEC:02
METEHAN KAYA	21401258	SEC:01

0. WHAT WE DID SO FAR

While we were implementing the project, our primary aim was to provide the users easily understandable syntax for propositional calculus. However, since we have not familiar with the Lex and Yacc tool before, we encounter some undesirable issues. During the implementation of the first part of the project we mastered Lex tool.

Our biggest challenge was to keep our grammar consistent and unambiguous. During the implementation of project since there are numerous operations, primitive types and objects, our lex file give several warnings even if the file compile smoothly. The main reason behind these facts was the complexity of our grammar. Since we declared many rules for our language, we experienced some problems because of the order of the statements. In order to handle it, we made small changes on our grammar.

After making changes on our grammar, we spent the some time on solving reduce/reduce and shift/reduce conflicts. In order to solve this issue, we emphasised on performing more tests and we mostly traced the grammar step by step.

In addition to this, to give information to user about acceptance of the input, the parser prints out a message. If the input is valid, the parser shows “Input program accepted.”. Otherwise, the parser gives an error message like "*-syntax error”. And, * indicates the line number of source code that contains the error.

1.PACK007 TUTORIAL

PACK007 Tutorial is a comprehensive tutorial which helps you to understand PACK007 basics with the help of sample code segments.This tutorial is even for anybody who does not know anything in any programming language.

2. WHAT IS PACK007?

Pack007 is a language providing a very easy syntax for propositional calculus. The main aim is to have an easy to use syntax with a familiarity of the current languages to provide a smoother adaptation phase for programmers. Pack007 includes all of the basic functionalities you look in a programming language and it also has some specific parts to make programmers’ life for propositional calculus.

3.COMMENTS

The comments are statements that are not executed by compiler. The comments are used to provide information about statements, functions, variables etc. PACK007 supports single line comment that is used to comment for only one line. The characters come after the "??" will be treated as a comment.

```
??This line is a comment.
```

Example comment line is following;

```
Double £getArea() {
 ??the area of square will be calculated here
}
```

4. GENERAL CONCEPTS

start

start: library_list global_variable_list main_definition method_list { printf("Input program accepted. \n"); }

Main composed of 4 structures:

```
Library BINARYTREE

Final SIZE = 100;
Integer value;

void £changeValue(Integer newValue) {
        value = newValue;
}

Main() {
        ??Main’s body
}
```

main_definition

main_definition: MAIN LP RP LB void_method_body RB ;

It defines Main’s structure:

```
Main() {
$calcCube();
}
```

Defining Library
library_list

```
library_list:    library_definition
                 | library_definition library_list ;
```

At the beginning of the file, there are some libraries:

```
Library NODE
Library EDGE
Library CALCULUS
```

library_definition

```
library_definition: LIBRARY LIBRARYNAME ;
```

It defines a single library:

```
Library BINARYTREE
```

5.VARIABLE TYPES AND DECLARATION

The primitive types and PACKAGE are considered to variable types.

variable_type

```
variable_type: primitive_type
               | PACKAGE ;
```

5.1. Primitive Types

Here is the BNF description of primitive types in PACK007:

primitive_type

```
primitive_type:INTEGERTYPE
              | DOUBLETYPE
              | STRINGTYPE
              | BOOLEANTYPE ;
```

PACK007 supports four primitive types that are Integer, Double, String and Boolean.

Integer is to represent integers which is composed of one or more digit.

```
    Integer num1;
    number = 3;
```

Double is to represent double number which is composed of integer and decimal point.

```
    Double average;
average = 4.0;
```

String is to represent strings which is considered as combination of characters.

```
    String str1;
    str1 = "Merhaba";
```

Boolean is to represent boolean values which return true or false.

```
    Boolean flag;
    flag = True;
```

final_variable_declaration

```
final_variable_declaration:    FINAL variable_type assign ;
```

It defines a final variable’s declaration:

```
Final Double pi = 3.14;
```

non_final_variable_declaration

```
non_final_variable_declaration: variable_type assign
                              | complex_variable_declaration ;
```

A non-final variable's declaration can be stated in 2 different ways:

1. Integer value = 5;
2. Integer value;

simple_variable_declaration

```
simple_variable_declaration:
        variable_type variable_list
        ;
```

It defines a simple variable declaration where there is no value assigned:

Integer value;

global_variable_list

```
global_variable_list:
                        | final_variable_declaration SEMICOLON global_variable_list
                        | non_final_variable_declaration SEMICOLON global_variable_list ;
```

final_variable_declaration

```
final_variable_declaration:    FINAL variable_type assign ;
```

non_final_variable_declaration

```
non_final_variable_declaration: variable_type assign
                                | complex_variable_declaration ;
```

complex_variable_declaration

```
complex_variable_declaration:       variable_type variable_list ;
```

Different types of declarations can follow each other:

```
Final Integer SIZE = 1000;
String word = "Pack007"
String sentence;
```

variable_list

```
variable_list:
        VARIABLE COMMA variable_list
        | VARIABLE
```

;

This is a list that contains variable names:

value , size , length

5.2. Predicates

In order to implement any predicate, we should know the declaration of it. First of all, the return type of predicate should be either true or false. These return type should be kept in the predicate variable. In order to create a predicate variable user should use “Predicate” keyword. After this key word, you should pick any string in order to keep the return type of predicate. Right hand side of the equation should be contains a function(PREDICATENAME). This function has at least two parameter.The type of parameters of this function should be any explicit value(integer value, string value, double value or boolean value)

common_method_stmt

```
common_method_stmt:       COMMENT
                          | assign SEMICOLON
                          | array_assign SEMICOLON
                          | void_method_call SEMICOLON
                          | input_definition SEMICOLON
                          | output_definition SEMICOLON
                          | final_variable_declaration SEMICOLON
                          | non_final_variable_declaration SEMICOLON
                          | ARRAYTYPE ARRAYNAME ASSIGNMENT INITIALIZE SEMICOLON
                          | PREDICATE VARIABLE ASSIGNMENT PREDICATENAME LP
call_list RP SEMICOLON ;
```

call_list

```
call_list:          VARIABLE COMMA call_list
                    | explicit_value COMMA call_list
                    | array_value COMMA call_list
                    | VARIABLE
                    | explicit_value
                    | array_value ;
```

The examples as following:

```
Predicate p1= €isSame(groupId,10);
Predicate p2= €isEqual(groupName, “PACK007”);
```

5.3 Arrays

In order to implement array, the declaration of the array should be known. Firstly, return type of array can be changed according to the input given by user. The return type of array should be kept in array name that should be determined by user. In order to create array name user should use “Array” keyword. After this keyword, user should use “@” symbol before the array name that is chosen. Right hand side of equation should be consist of “Initialise()”. After declare the array, user may give any string, integer or double value in the array. In order to fill the array, right square bracket, left square bracket and the index that will keep the given value should be use.

```
common_method_stmt:  array_assign SEMICOLON| ARRAYTYPE ARRAYNAME ASSIGNMENT
INITIALIZE SEMICOLON

array_assign:  array_value assignment_operation explicit_value
               | array_value assignment_operation VARIABLE ;

array_value:   ARRAYNAME LS explicit_value RS
                              | ARRAYNAME LS VARIABLE RS ;
```

```
??Example Code Segment
Array @fact = Initialize();
@fact[0] = 0;
Final Integer maxn = 1000;
For( i = 1 ; i <= maxn ; i += 1 ) {
 mult *= i;
 mult %= primeMod;
 @fact[i] = mult;
}
```

6.Methods

In order to declare a method, first of all, the return type should be decided.There are two method definitions in PACK007. One is void_method_definition and other is returnable_method_definition. If the return type void the method does not return any value. If the method is returnable that means returns a variable_type (Integer,Boolean,Double and String).

6.1. Method Definition

In PACK007, there are two types of function definitions.

void_method_definition

void_method_definition: VOID VMETHODNAME LP parameter_list RP LB void_method_body RB ;

returnable_method_definition

returnable_method_definition: variable_type RMETHODNAME LP parameter_list RP LB returnable_method_body RB ;

??Example Code Segment

```
Integer £calculateSum(Integer a, Integer b){
        Integer sum=a+b;
        Return sum;
}
```

??Example Code Segment

```
Void $swap(Integer a,Integer b){
        Integer c=a;
        a=b;
        b=c;
}
```

6.2. Method Body

```
void_method_body:
                        | void_method_stmt void_method_body ;
```

Contains statements.

```
returnable_method_body:
                                | returnable_method_stmt returnable_method_body ;
```

Contains statements.

6.3. Method Statement

void_method_stmt

```
void_method_stmt:    void_loop
                     | void_conditional
                     | common_method_stmt
```

It contains common methods for void and returnable methods.

returnable_method_stmt

```
returnable_method_stmt:     returnable_loop
                            | returnable_conditional
                            | return_definition SEMICOLON
                            | common_method_stmt ;
```

It contains common methods for void and returnable methods. In addition, rules may take return definitions.

common_method_stmt

```
common_method_stmt:         COMMENT
                            | assign SEMICOLON
                            | array_assign SEMICOLON
                            | void_method_call SEMICOLON
                            | input_definition SEMICOLON
                            | output_definition SEMICOLON
                            | final_variable_declaration SEMICOLON
                            | non_final_variable_declaration SEMICOLON
                            | ARRAYTYPE ARRAYNAME ASSIGNMENT INITIALIZE SEMICOLON
                            | PREDICATE VARIABLE ASSIGNMENT PREDICATENAME LP
call_list RP SEMICOLON ;
```

Case 1: ?? This is a comment line
Case 2: var = True;
Case 3: @hash[“Group”] = 7;
Case 4: $func(number);

Case 5: Input number;
Case 6: Output £func(1);
Case 7: Final Double pi = 3.14;
Case 8: Double pi = 3.14;
Case 9: Array @hash[] = Initialize();
Case 10: Predicate p1 = €isLarger(x, 5);

6.4. Parameter List

Functions can take no parameter, a parameter or parameter list.

parameter_list

```
parameter_list:
                | parameter_definition
                | parameter_definition COMMA parameter_list ;
```

Case 1: (Empty)
Case 2: Integer value
Case 3: Integer value , String word , Boolean var

parameter_definition

```
parameter_definition: variable_type VARIABLE ;
```

The examples are as following:

Double radius
Integer a

7.Conditional Statements and Loops

Before getting start, we need to clarify what the the condition and compare_opeation mean. The BNF description of condition is as following:

compare_operation

```
compare_operation:  ISEQUAL
                    | NOTEQUAL
                    | LESS
                    | LESSOREQUAL
                    | GREATER
```

```
                        | GREATEROREQUAL ;
```

There are 6 kinds of compare operations: == , != , < , <= , > , >=

condition

```
condition:    connective_list
              | VARIABLE compare_operation VARIABLE
              | VARIABLE compare_operation explicit_value
              | explicit_value compare_operation VARIABLE
              | explicit_value compare_operation explicit_value ;
```

A condition can be defined in 6 different ways:

```
    true (or false)
    true | false
    var1 < var2
    var1 < 5
    5 < var1
    5 < 10
```

It is a list of connectives where many connectives can be defined.

connective_list

```
connective_list:    connective_value connective_operation connective_list
              | connective_value ;
```

Connective and compare operations are important operations in PACK007. "compare_operation" is used to compare the variables and constants. The BNF description of it is like following:

connective_operation

```
connective_operation:    AND
                         | OR
                         | IMPLY
                         | IFF ;
```

There are 5 different connective operations: | , & , -> , <->

connective_value

```
connective_value:    VARIABLE
                     | BOOLEANVALUE
```

```
                    | VALUENOT ;
```

$Example Code Segment

```
    Boolean x=true;
    Boolean y=false;

    Boolean flag1=x&y;
    Boolean flag2=x~y;
    Boolean flag3=x->y;
```

7.1. Conditional Statements

In PACK007 conditional statements are following:

- **void_conditional**
- void_if_conditional
- void_if_else_conditional

- **returnable_conditional**
- returnable_if_conditional
- returnable_if_else_conditional

```
void_conditional:       void_if_conditional
                        | void_if_else_conditional ;


returnable_conditional: returnable_if_conditional
                         | returnable_if_else_conditional ;
```

If Statements

The BNF description of condition is void_if_conditional as following:

- **void_if_conditional**

void_if_conditional: IF LP condition RP LB void_method_body RB ;

??Example Code Segment

```
Void $playWeird( Boolean var1 , Boolean var2 , Boolean var3 ) {
        If( var1 & true | ~var2 -> ~false <-> var3 ) {
                Output "Condition 1";
        }
}
```

- **returnable_if_conditional**

returnable_if_conditional: IF LP condition RP LB returnable_method_body RB ;

The statements inside If are executed until the condition is true. If the given condition is not satisfied, then If is skipped and the set of code comes after "}" is executed. A simple example of If-loop in PACK007 as following:

??Example code segment

```
Double £getArea( Double radius ) {
Double radius, area;
        If( radius <= 0 ) {
         Return 0;
        }
}
```

If-Else Statements

The BNF description of If-Else condition is as following:

- **void_if_else_conditional**

void_if_else_conditional: IF LP condition RP LB void_method_body RB ELSE LB void_method_body RB ;

?? Example Code Segment

```
    Void $playWeird( Boolean var1 , Boolean var2 , Boolean var3 ) {
        If( var1 & true | ~var2 -> ~false <-> var3 ) {
            Output "Condition 1";
        }
        Else {
            Output "Condition 2";
        }
    }
```

- **returnable_if_else_conditional**

returnable_if_else_conditional: IF LP condition RP LB returnable_method_body RB ELSE LB returnable_method_body RB ;

??Example code segment

```
    Double £getArea( Double radius ) {
    Double radius, area;
        If( radius <= 0 ) {
            Return 0;
        }
        Else {
         area = pi;
         area *= radius;
         area *= radius;
         Return area;
        }
    }
```

7.2. Loops

Loops are useful in programming when the block of code will be executed many times. PACK007 supports two types of loops that are void_loop and returnable_loop. The BNF description of them are as following:

void_loop

```
void_loop:      void_for
                | void_while ;
```

returnable_loop

```
returnable_loop:        returnable_for
                        | returnable_while ;
```

While Loop

"While" is used as a keyword to initialise While-loop. The statements in While-loop are executed as long as given condition is true. When the condition is false, While-loop is skipped and the rest of the code is executed.

The BNF description of While loop is as following:

- **void_while**

```
void_while:     WHILE LP condition RP LB void_method_body RB ;
```

??Example Code Segment

```
Void $printStarTriangle() {
            Integer i, j, n;
            Input n;
            i = 0;
           While( i < n ) {
                      j = 0;
                      i += 1;
                      While( j < i ) {
                        j += 1;
                       Output "*";
                       }
           }
}
```

- **returnable_while**

```
returnable_while:       WHILE LP condition RP LB returnable_method_body RB ;
```

The example of While-loop is as following:

??Example code segment

```
Integer number=0;

While (number<100){
        $print(number);
        number=number+1;
}
```

For Loop

"For" is a keyword to initialize For-loop. As shown above, between the "(" and ")" the initial value is initiliazed and the statements in for loop is executed until the condition becomes false. When the condition false, For-loop is skipped and the rest of the code is executed.
The BNF description of For Loop as following:

- **void_for**

void_for: FOR LP assign SEMICOLON condition SEMICOLON assign RP LB void_method_body RB ;

- **returnable_for**

returnable_for:FOR LP assign SEMICOLON condition SEMICOLON assign RP LB returnable_method_body RB ;

To give an example For-loop in PACK007 as following,

??Example code segment

```
Integer number;
For (number=0 ; number<100 ; number=number+2){
        $print(number);
}
```

method_list

```
method_list:
                | void_method_definition method_list
                | returnable_method_definition method_list ;
```

There can be many methods above the Main:

```
Void $changeValue() {}
Integer £getValue(Integer value) { Return value; }
```

input_output_definition

```
input_definition:       INPUT variable_list ;



output_definition:      OUTPUT call_list
                        | OUTPUT returnable_method_call ;
```

Inputs and outputs are also possible in Main:

```
Input length , value;
Output size , height;
```

assign

```
assign: VARIABLE assignment_operation VARIABLE
        | VARIABLE assignment_operation explicit_value
        | VARIABLE assignment_operation returnable_method_call ;
```

return_definition

```
return_definition:      RETURN VARIABLE
                        | RETURN explicit_value ;
```

There are 2 different ways a return statement can be defined:

1. VARIABLE: Return value;
2. explicit_value: Return 10.5;

explicit_value

```
explicit_value: INTEGERVALUE
              | DOUBLEVALUE
              | STRINGVALUE
              | BOOLEANVALUE ;
```

assignment_operation

```
assignment_operation:      ASSIGNMENT
                    | PLUSASSIGNMENT
                    | MINUSASSIGNMENT
                    | MULTASSIGNMENT
                    | DIVASSIGNMENT
                    | MODASSIGNMENT ;
```

There are 6 types of assignments:

1. ASSIGNMENT: var = 5;
2. PLUSASSIGNMENT: var += 5;
3. MINUSASSIGNMENT: var -= 5;
4. MULTASSIGNMENT: var *= 5;
5. DIVASSIGNMENT: var /= 5;

6: MODASSIGNMENT: var %= 5;