CS 315 Programming Languages

Project 2 - Report GROUP 11

Panda Language

Melike Arslan	21601025	Section 2
Berk Güler	21402268	Section 2
Gökçe Sefa	21400596	Section 2

Table of Contents

Part A - Language Design

Introduction

Panda Language is a new language to calculate propositional calculus then tell users what is true or false. In this project, new language of Panda includes language design of BNF (Backus- Naur Form), Lexical analysis and example program in order to specify the new language features in detailed. Some list of features are required in project description and we added some different features to enrich language properties.

Complete BNF Of Panda Language

```
<program>                 ::= go <main_declaration> finish
                          | go <statements> finish

<main_declaration>        ::= pandaMain <LP> <RP> { <statements> }

<statements>              ::= <statements> ; <statement>
                    | <statement>

<statement>               ::= <control_statement>
                    | <loop_statement>
                    | <regular_statement>
                    | <predicate_definition>
                    | <empty>

<control_statement>            ::= <with_else>
                          | <without_else>

<with_else>               ::= if <LP><proposition><RP> <with_else> ifn’t < with_else >
                    | {<loop_statement>}
                    | {<regular_statement>}

<without_else>            ::= if <LP><proposition><RP><statements>
                    | if<LP><proposition><RP> <with_else> ifn’t <without_else>
```

```
<loop_statement> ::= cycle <LP> <proposition> <RP>{ <statements>}


<regular_statement>                    ::=  <return_statement>
                                       | <assignment_operation>
                                       | <input>
                                       | <output>


<predicate_definition>                 ::= predicate <predicate_name> <LP> <param_list>
                                  <RP>{ <statements>}


<predicate_instantiation> ::= $<predicate_name> <LP> <param_list> <RP>


<predicate_name> ::=  <predicate_name><digit>
                      |<predicate_name><letter>
                      | <letter>
<proposition>            ::= <proposition> <Equivalence_operator> <imply_expression>

                       | <imply_expression>

<return_statement>              ::= return <identifier>
                                | return <truth_val>


<assignment_operation > ::= <identifier> <assign_op> <proposition>

                               | <proposition> <assign_op> <truth_val>

                              | <proposition> <assign_op> <char>

                              | <proposition> <assign_op> <characters>

                              | <proposition> <assign_op> <integer>


<input >                 ::= input {<proposition>}

<output >                ::= output [<proposition>]
```

```
<param_list>              ::= <param_list>  <comma_op>  <identifier>
                         |<identifier>
                         |<integer>
                         |<truth_val>
                         | <empty>

<imply_expression> ::= <imply_expression> <implies_operator> <or_expression>
                           |<or_expression>

<or_expression>         ::= <or_expression><or_operator><and_expression>
                         | <and_expression>

<and_expression> ::= <and_expression><and_operator><not_expression>
                         | <not_expression>

<not_expression> ::= <not_operator><operand>
                         | <operand>

<logical_operators> ::= <and_operator>
                          | <or_operator>
                          | <not_operator>
                          | <implies_operator>
                          | <equivalence_operator>

 <operand>             ::= <identifier>
                        | <predicate_instantiation>

<constants>               ::= readonly <identifier> <assign_op> <integer>
                           | readonly <identifier> <assign_op> <truth_val>

<identifier>              ::= <identifier> <integer>
                         | <identifier> <letter>
```

```
                          | <letter>
                          | <empty>

 <string>                ::= “<characters>”

<characters>            ::= <characters> <letter>
                          | <characters> <symbol>
                          | <characters> <digit>
                          | <letter>
                          | <symbol>
                          | <digit>

<char>          ::= ‘<letter>’
                  | ‘<symbol>’
                  | ‘<digit>’

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

 <truth_val> ::= true
                  | false
                  | unknown

 <empty>                 ::=
```

Explanation for Each Language Construct

Program <program>:

A Program is a non-terminal that consists of statements. Our new language, Panda, have a main method in the project 2, so this non-terminal contains main program procedure and statements. It has to be declared between **go** and **finish** reserved words to specify the beginning and the end of the program. Panda Language enable users to execute different statements and main method in complete program.

Statements <statements>:

Statements, which is also a non-terminal, can consist of a set of statements, or it can can consist of a single statement and lastly it can be empty, which denoted an empty program. This non-terminal does not require any reserved words in the beginning or the end unlike the <program>. Each statements have various and different responsibilities in order to perform different actions.

Statement <statement>:

A <statement> can be either one of these: A control statement <control_statement>, a loop statement <loop_statement>, a regular statement <regular_statement> or a predicate definition <predicate_definition> that are separated by a ‘;’ (semicolon) which enable readability to program. The reason it can be either one of those is because each type of statement has different rules. This non-terminal can allow Statements to work in proper program flow with their unique actions.

Control Statement <control_statement> :

Control Statement is a conditional statement in our program. A Control Statement can either be an if statement not followed by an else statement or an if statement followed by an else statement. These can also be identified as matched or unmatched statements. If an ‘if’ statement not followed by ‘else’ is defined, it means that ‘if’ statement is unmatched and rules different than the ‘if’ followed by an ‘else’, called unmatched, apply to it. This non-terminal is a recursive one to allow for more nested and simply more control statements in program. Matched and unmatched difference can improve reliability to the program.

Loop Statement <loop_statement>:

A Loop Statement consists of a proposition in between parenthesis and continues with a set of statements. It has to start with the reserved word **cycle**, which is a substitute for the while loop. This non-terminal takes a <proposition> as a parameter which can be evaluated to truth values which determines whether the statements in the **cycle** should be executed, and if so for how many iterations. Also project 2 has curly bracket to separate the statements to gain readability.

Regular Statement<regular_statement>:

A regular statement is a statement other than a control statement, a loop statement or a predicate definition which includes return statements <return_statement>, assignment operations <assignment_operation>, inputs<input> and outputs <output>.

Predicate Definition <predicate_definition>:

A Predicate Definition is an another non-terminal and corresponds to a function call that returns a truth value which indicates that it resembles a boolean returning function definition. It starts with the reserved word **predicate** and consists of the predicate name and a list of parameters in between parentheses. Predicate Definition has statements after the parameter list. Statements depends on the parameters which can be defined as true or false in advance. Thus, statements can return true or false depending on the parameters' truth values. Also project 2 has curly bracket to separate the statements to gain readability.

With_else <with_else>:

This is a non-terminal and matched control statement which includes regular statements, loop statements and it also stands for an if statement. This if statement starts with the reserved word **if** continued by a logical expression in between parentheses followed by another <with_else> and reserved word **ifn't** also followed by another <with_else>. This recursive <with_else> denotes that in an if statement, there can be another if statement. We shorten if not as a ifn't instead of else. Also second iteration of the project has curly bracket to separate the statements to gain readability for loop_statement, regular_statement.

Without_else <without_else>:

This is a non-terminal and unmatched control statement which means that there is an imbalance with the if statements, there is an if without an else, in our case there is an **if** without an **ifn't**.

Proposition <proposition>:

Propositions can have one of the truth values such as true or false. Proposition includes recursive logical expressions, expressions and comparisons of these expressions by using *if and only if* (<==>) and logical operators. Proposition is an important part of the program because statements return true or false depend on these propositions' truth value. Propositions can be used in control statement, regular statements and assignment operation in the program. That is, almost all statements need propositions to specify their activity. Truth values for propositions can be defined once such as constants or assignment operator. Thus, we can reuse these propositions in different statements without any changes.

Return statement <return_statement>:

This kind of statement is for returning either variables or truth values from functions in our case predicates. It starts with the reserved word **return**.

Assignment operation <assignment_operation >:

This is a non-terminal indicates values such as propositions, identifier, strings and truth values with assignment operator <assign_op> which consists of two equal signs(==) in our Panda Language.

Input <input >:

Starts with the reserved word **input** and is followed by a proposition in between curly brackets to distinguish between input and output in the program to gain readability.

Output <output>:

Starts with the reserved word **output** and is followed by a proposition in between square brackets to gain readability.

Predicate Name <predicate_name>:

The name of the specified predicate definition or predicate instantiation. Every predicate name is not necessarily unique since predicate definition may differ with different parameter lists such empty or two identifiers.

Predicate Instantiation <predicate_instantiation>:

Predicate instantiation is a non-terminal and different from the predicate definition in not using statements. It has predicate name and parameter list. Predicate instantiation also maynot be unique because of using different parameter lists which can have two parameters, one parameter or empty options in Panda Language. Because of that situation, although the predicate name is same to the other predicate instantiation or definition, if parameter list number different, return value also can be different.

Empty <empty>:

Empty is a non-terminal and it helps to show that some non-terminals may be empty. In Panda Language; Statements<statements>, parameter list <param_list> may have empty.

Parameter List <param_list>:

This denotes the parameter list for each predicate definition and also for each predicate instantiation. If it denotes multiple parameters then they are separated by ‘,’ (comma). Also it has different variances as a parameter because identifiers, integers, truth values and empty option can be used in parameter list. Also, statements’ return value depends on parameter lists’ truth values according to propositions.

Imply Expression <imply_expression>:

This non-terminal is expressed with another imply expression with an implies operator and an or expression or it is expressed only with an or expression. An imply expression results with either true, false or unknown. For example, if the proposition before the operator is true and the proposition after the operation is true, then the result is true. If the proposition before the operator is false and the proposition after the operation is false, the result is also true.

Or Expression <or_expression>:

Or expression has either true or false values. If the proposition is true, it value is true but If the proposition truth value is false and it is false. Or expression is also recursive an it includes and expressions in our language grammar. Or expression uses <or_operator> which is | in Panda language.

And Expression <and_expression>:

And expression is also recursive expression because it has also not expression which has operands that is a main compounds of expressions. And expression uses and operator as & to connect with propositions, identifiers or predicate instantiations. There is only one true value possible, if both left and right side of expression are true then final value is true. Otherwise it returns false value.

Not expression <not_expression>:

Not expression makes expressions truth value is an opposite truth value in expressions. It uses (!) operator to specify not expressions in the language. This expression reverses the value.

Operand <operand>:

Operand is an another computer instruction to specify that the data features and activities and includes wide range of expressions. We use operand especially generalize operations such as identifiers, propositions and predicate instantiations to be used in expressions directly or recursively.

Truth values <truth_val>:

This non-terminal includes true, false and unknown terminals. These terminals are generally return values of predicate instantiations.

Identifiers <identifier>:

Identifiers consist of another identifier with either an integer or a string, or they can consist of only a string. This denotes that an identifier can't start with an integer.

Constant <constants>:

Constants define the unchangeable identifiers, requires the **readonly** keyword in front of the identifier name.

Assignment operator <assign_op>:

This non-terminal is the operator for <assignment_operation>. It is denoted by "==". It is used for assigning either a string or a proposition to an identifier.

New Line <newLine>:

This denotes a new-line ‘\n’.

Left Parenthesis <LP>:

This denotes the left paranthesis, “(“.

Right Parenthesis <RP>:

This denotes the right paranthesis, “)“.

Tokens Of Panda Language

1. Reserved Words

Go ::= go reserverd word declares that statements are starting **Finish** ::=
finish reserved word decleras that statements are finished **If** ::=
conditional statement starts with if

Ifn’t ::= inverse of if in a conditional statement

Cycle::= Loop, resembles the while loop in other languages

Predicate ::= keyword necessary to declare predicate definition

Return::= used in return statements to “return” a truth value or an identifier

Input ::= input received from the user **Output**
::= output received from the program **True::=**
one of the truth values

False::= one of the truth values

Unknown::= one of the truth values

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

```
<comma_op>            ::= ,
                       | <empty>
<assign_op> ::=       ==

<newLine> ::=         \n

<symbol>              ::=? | @ | [ | \ | ] | ^ | _ | ` | { | }| . | |

<and_operator>                ::= &

<or_operator>                 ::= |

<not_operator>                ::= !

<implies_operator>                    ::= ==>

<equivalence_operator> ::= <==>

<LP>                          ::= (

<RP>                          ::= )

<LCB>                         ::= {

<RCB>                         ::= }

<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
```

2. Comments

Comments are placed in between two squares (#). It's a crucial part of every programming language as it can be considered as a communication tool between programmers. Gives information about the implementation of functions, variables etc.

3. Identifiers

Identifiers, as the name suggests are indicated as <identifiers> in our Panda language. Similar to other languages in order to increase the readability, identifiers cannot start with a digit and has to start with a letter. However, an identifier can have a digit in second or other spaces only excluding the first character. Identifiers can hold the values of string or integer.

Part B
Lexical Analysis Revised

```
%x PREDICATE

%{

int lineCount = 1;

%}

letter [a-zA-Z]

integer [0-9]

symbols [\?\@\[\]\\\^\_\}\{\']

alphanumeric ({letter}|{integer}|{symbols})

%%

\n {

        lineCount++;

}

go { return (PROG_BEGIN); }

finish { return (PROG_END); }

pandaMain { return (MAIN); }

if { return (IF); }

ifn't { return (ELSE); }

cycle { return (WHILE); }

readonly { return (CONSTANT); }

return { return (RETURN); }

predicate { return (PREDICATE); BEGIN (PREDICATE); }

<PREDICATE>[ \t]{letter}{alphanumeric}* { return (PREDICATE_NAME); BEGIN (INITIAL); }

\${letter}{alphanumeric}* { return (PREDICATE_NAME); }

input { return (INPUT); }
```

```
output { return (OUTPUT); }

true { return (TRUE); }

false { return (FALSE); }

unknown { return (UNKNOWN); }

{letter}{alphanumeric}* { return (IDENTIFIER); }

\#.*\# { return (COMMENT); }

{integer} { return (INT); }

{letter}? { return (LETTER); }

\".*\" { return (STRING); }

\'.?\'  { return (CHAR); }

\=\= { return (ASSIGN_OP); }

\=\=\=  { return (EQUALITY_OP); }

\( { return (LP); }

\) { return (RP); }

\{ { return (LCB); }

\} { return (RCB); }

\[  { return (LSQB); }

\]  { return (RSQB); }

\; { return (SEMICOLON); }

\| { return (OR_OP); }

\$ { return(DOLAR);}

\& { return (AND_OP); }

\!  { return (NOT_OP); }

\, { return (CM); }

\<\=\=\> { return (EQUIVALENCE_OP); }

\=\=\> { return (IMPLIES_OP); }

%%

int yywrap() { return 1; }
```

YACC

```
%{

#include <stdio.h>

int yylex();

int yyerror();

%}

%start start

%token PROG_BEGIN PROG_END MAIN IF ELSE WHILE CONSTANT

RETURN PREDICATE PREDICATE_NAME INPUT OUTPUT TRUE FALSE

UNKNOWN IDENTIFIER COMMENT INT STRING CHAR ASSIGN_OP

EQUALITY_OP NEWLINE LP RP LCB RCB LSQB RSQB SEMICOLON

OR_OP AND_OP NOT_OP CM EQUIVALENCE_OP IMPLIES_OP LETTER

%%

start: program ;

program: PROG_BEGIN statements PROG_END

 | PROG_BEGIN main_declaration PROG_END ;

main_declaration: MAIN LP RP LCB statements RCB ;

statements: statements SEMICOLON statement | statement | statements

SEMICOLON COMMENT;

statement: control_statement

 | loop_statement

 | regular_statement

 | predicate_definition

 | empty ;
```

control_statement: with_else

| without_else ;

with_else: IF LP proposition RP LCB with_else RCB ELSE LCB with_else RCB

| LCB loop_statement RCB

| LCB regular_statement RCB;

without_else: IF LP proposition RP LCB statements RCB

| IF LP proposition RP LCB with_else RCB ELSE LCB without_else RCB ;

loop_statement: WHILE LP proposition RP LCB statements RCB ;

regular_statement: return_statement

| assignment_operation

| input

| output ;

predicate_definition: PREDICATE predicate_name LP param_list RP LCB statements RCB ;

predicate_instantiation: '$'PREDICATE_NAME LP param_list RP ;

predicate_name: predicate_name INT

| predicate_name LETTER

| LETTER ;

proposition: proposition EQUIVALENCE_OP imply_expression | imply_expression ;

return_statement: RETURN IDENTIFIER

| RETURN truth_val ;

```
assignment_operation : IDENTIFIER ASSIGN_OP proposition
    | IDENTIFIER ASSIGN_OP STRING
 | IDENTIFIER ASSIGN_OP CHAR
 | IDENTIFIER ASSIGN_OP INT
   | proposition ASSIGN_OP truth_val
 | CONSTANT IDENTIFIER ASSIGN_OP INT
 | CONSTANT IDENTIFIER ASSIGN_OP truth_val ;
input: INPUT LCB proposition RCB ;
output: OUTPUT LSQB proposition RSQB ;
param_list: param_list CM IDENTIFIER
 | IDENTIFIER
 | INT
 | truth_val
 | empty ;
imply_expression: imply_expression IMPLIES_OP or_expression
  |or_expression ;
or_expression: or_expression OR_OP and_expression
  | and_expression ;
and_expression: and_expression AND_OP not_expression
  | not_expression ;
not_expression: NOT_OP operand
  | operand ;
operand : IDENTIFIER
```

```
 | predicate_instantiation ;

truth_val: TRUE | FALSE | UNKNOWN ;

empty: ;

%%

#include "lex.yy.c"

extern int lineCount;

int yyerror(char *s) {

 printf("SYNTAX ERROR ON LINE %d!\n", lineCount);

}

int main() {

 if (yyparse() == 0)

  printf("Input program is valid.\n");

 return 0;

}
```

Part C - Example Programs

Example 1:

```
go
          predicate berkGetsCoffee(heHasMoney, heIsTired)
          { getsCoffee == heHasMoney & heIsTired;
          if (getsCoffee)
                    return true;

}
finish

ifn't
```

```
return false;
```

Example 2:

```
go cycle
{




# This is a test program # input inin;
output outout;
array [5] == gokce;

readonly classYear == 3;
        studentName == "Melike Arslan";
        studentGroup == 'A';

        if( !melikeStudiesForExam & melikeIsSick)
        {


        }
        ifn't{


        }

 return failExam;
 out == !failExam | getHealthReport; condition == unknown;


                          predicate isPracticing(question1,
                          2question){ return false;


}
finish

}
```