CS315
PROGRAMMING LANGUAGES

Project-1

Design and Lexical Analysis of a Propositional Logic Programming Language

Group 16

Bora Ecer	21501757	Section: 02
Alp Ege Baştürk	21501267	Section:02
Deniz Alkışlar	21502930	Section: 02

Name of the Language: pclog

1. Syntax, Definitions and Conventions

The syntax of the language is similar to C descendant languages.

Keywords for if-else statements is “if” and “else”. Language allows if statement without else statement. Else is matched to the nearest unmatched if statement above the else statement. If statement requires condition to be written inside parentheses following the “if” keyword.

Keywords for loops are similar to C descendant languages. There are three loop types; *for, while* and *do-while.* Use of these loops are similar to the C type languages. While loop requires the condition to be defined in parentheses following the “while” keyword. For requires declaration, condition and update rule to be defined in parentheses. Body of the loops can be inside braces “{“ “}” or not. This is similar to the C type languages. Do-while requires one *do* block, followed by a while keyword followed by the condition in parentheses and a semicolon.

Language provides two types, integer defined with keyword “int” and boolean defined with “boolean”. Each variable must be declared with its type before its use in a program.

Predicates and functions are defined as return type followed by the name followed by parameters list inside parentheses followed by braces. These are called as name followed by parameter list inside parentheses.

Examples:

```
int foo ( int a )        int x = 0;
{                        foo ( x ) ;
        ...
}
```

```
boolean q = false;          boolean q = false;
boolean p = true;           boolean p = true;
If ( p )                    If ( p ) {
        q = true;                   q = true;
                                    p = false;
                            }
```

2. BNF description of the Language

```
 S →   <statements>
<statements> → <statements>
                   | <statements> <statement>
<statement> → <non_if_statement>
                   | <if_statement>

<non_if_statement> →  <proposition> | <complex_statement>
                            | <function_call>
                            | <predicate_call>
                            | <do_while_loop>
                            |  <for_loop>
                            |  <while_loop>
                            |  <logical_expr>
                            | <function>
                            | <predicate>

<proposition> → <boolean_expr>
                   | <identifier>

<complex_statement> →  "(" <statement> ")"
                                 | <statement> <logical_operator> <statement>
                                 | "!" <statement> ;

<predicate> →  <predicate_name> <arg_list> { <statements> }
<arg_list>  →  ( term_list )
<term_list> →  <term>
          | <term_list> , <term>

<term> →  <identifier>
            | <const>

<predicate_name> →  <identifier>
                            | <identifier> <identifier>

<function>  →  <function_name> ( <arg_list> ) { statements }
<function_name> →  <alphabetic>

<alphabetic> →  <term_list>
           |  <term_list> <term>

<function_call> → <function_name> ( <arg_list> );
```

```
<predicate_call> → <predicate_name> ( <arg_list> );

// Aritmethic
<arithmetic_assignment> →  <identifier> assignment_operator <arithmetic_expr> ;

<arithmetic_expr> →   <identifier> <arithmetic_operator> “(“ <arithmetic_expr> “)”
                      | <identifier>



// Logical
<logical_assignment> →  <identifier> assignment_operator <logical_expr> ;

<logical_operation> →   <identifier> <logical_operator> <logical_expr> ;


// Loops
<while_loop> →  while ( <logical_expr>) <statements>
                    | while (<logical_expr>) { <statements> }

<do_while_loop> →  do <statements> while ( <logical_expr> );
                         |  do { <statements> } while ( <logical_expr> );

<for_loop> →  for( <logical_assignment>; <logical_operation>; <logical_assignment>)
{<statements>}
                | for( <logical_assignment>; <logical_operation>;
<logical_assignment>)<statements>

// I/O
<input_statement> → input “(“ <identifier> ”)”

<output_statement> →  output “(“ <identifier> “)” ;

// If-Else && If-Else-Then
<if_statement> →   <matched_if>
                      | <unmatched_if>

<matched_if> →   if(<logical_expr>) <matched_if> else <matched_if>
                    | <non_if_statement>

<unmatched_if> →  if(<logical_expr>) <statements>
                        |  if(<logical_expr>) <matched_if> else <unmatched_if>

//Operators
assignment_operator →  "="
<arithmetic_operator>    →  "*"
```

```
                        |  "/"
                        |  "+"
                        |  "-"
<logical_operator>   →  "&&"
                        |  "||"
                        |  "!="
                        |  "=="
                        |  "=>"
                        |  “=<”
                        | “-->”
//Other declerations
<boolean_expr> →  "true"
                  |   "false"

<integer> → *1(sign) <positive_integer>

<positive_integer> → <digit>
                     |   <digit><positive_integer>


<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
sign → “-”
      |  “+”

<expr> → <boolean_expr>
        |   <integer>

input →  “read”
output →  “write”
input_location → “location”

<identifier>  →    <identifier>
               | <identifier>  a | ... | z | A | ... | Z

<variable_decleration> ->  <var>  <identifier> assignment_operator <expr>;
<array_decleration>  → <var> <identifier> [ <positive_integer> ] ;
<const_decleration>   →  const <var>  <identifier> assignment_operator <expr>;

<var> →  “int”
        | “boolean”

<logical_expr> →   <identifier> <logical_operator> “(“ <logical_expr> “)”
                | <identifier>
                | logical_not <identifier>

<comments> → # <identifier>
```

3. Details of each language construct (nonterminals)

- **<statements>:** A program in the language is made of statements. This can be one or more statements.
- **<statement>:** This is the basic meaningful block of the language which can be any meaningful block in the program. Statement can be any statement, proposition, complex statement, function or predicate call, loop, if and if-else or a logical expression.
- **<non_if_statement>:** Statements without if. Separation of the if allows rule for matching if and else statements in the grammar.
- **<proposition>:** A proposition can be a boolean expression which is True or False or p, q and r. This use will allow truth calculations in discrete mathematics.
- **<complex_statement>:** This statement is similar to the normal statement however it also allows use of parenthesis for a statement. Furthermore it allows logical comparison of multiple statements and negation of a statement
- **<predicate>:** Predicate is fundamentally a function call with boolean return value, with name and arguments.
- **<arg_list>:** This is the list of arguments to be passed to the call. It is a list of terms inside parentheses.
- **<term_list>:** This is a list of terms. It can be a one term or multiple terms.
- **<term>:** term can be an identifier or a constant. For example; **foo (boolean b) { <statements> }** is a predicate in the language. In this code snippet, foo is the predicate name boolean b is the argument list with boolean and b as identifiers. Statements inside the brackets are the body of the predicate.
- **<predicate_name>:** This is the name of the predicate which can be a string of characters defined by programmer.
- **<function>:** Is a similar structure to the predicate, however return value of it is not limited to {True,False}.
- **<function_name>:** This is the name of the function. It's consist of characters of the alphabet.
- **<alphabetic>:** Can be a term or list of terms. Since a term can be an identifier, which is a set of characters, it can be any string of characters.
- **<term_list>:** Is a list of terms separated by comma.
- **<term>:** Term can be an identifier or a constant.

- **<function_call>:** Means a call to a function with name and arguments list in parentheses.
- **<predicate_call>:** Means a call to a predicate with name and arguments list in parentheses.
- **<arithmetic_assignment>:** Allows assigning arithmetic values to variables. For example int x = 0; assignment.
- **<arithmetic_expression>:** Allows writing recursive arithmetic operations.
- **<logical_assignment> :** the intended usage of <logical_assignment> is assigning the resulting boolean value of a logical operation into another boolean. (e.g consider three booleans: a,b,c. a = (b && c)) This assignment will be useful for loops and if-else statements. <logical_assignment>. There is only one convention of <logical_assignment> which includes another non-terminal, named <logical_expr> which has conventions. Since <logical_expr> is common for most of the remaining non-terminals, they will be explained below in the report.
- **<logical_operation>:** logical operation is the computation of at least two boolean variables, using the basic logical operators. (e.g AND, OR, NEGATE, etc). <logical_operation> has three different associated conventions, which are <identifier>, <logical_operator> and <logical_expr>. So, it works just like <logical_assignment> but instead of an assignment, it just calculates the result. Th
- **<logical_expr>:** this is used to make recursively increasing logical operations, in both <logical_operation> and <logical_assignment>, the main reason this was added to make these non-terminals have recursive aspect. (e.g a = b && (c || (e == ….). This non-terminal also has three conventions.
    1. <logical_expr> → <identifier> <logical_operator> "(" <logical_expr> ")" which allows it to make nested logical operations.
    2. <logical_expr> → <identifier>, which makes it return only one identifier, if the operation is done using two boolean variables
    3. <logical_expr> → logical_not <identifier>, is to return the not value of the given boolean variable.
- **<while_loop>:** the non-terminal is used for the constructing while loops, which takes a logical expression as a parameter, and executes the statements given until the logical expression parameter is satisfied. There are two different conventions for this one

which are basicly the same but one of them has curly brackets {} around the <statements>, one of them does not.

- **<do_while_loop> :** Just like the while_loop, this non-terminal is used for constructing do-while loop, which executes in a similar way with while loop but it checks the parameter after the statements is executed. Again, it has two different conventions, just to add {} around the statements.
- **<for_loop> :** Is the non-terminal to construct the for loop, which takes three parameters in it, and executes the statements, until the given conditions are satisfied. Just like the other loops, it has two different conventions, just to add {} around the statements.
- **<input_statement> :** This non-terminal is for taking reading an input from a input_location, into a identifier.
- **<output_statement> :** This is the non-terminal for the printing the value inside a identifier to the screen.
- **<if_statement> :** the non terminal is used for the if-else selection. It has two different conventions which are <matched_if> and <unmatched_if>
- **<matched_if> :** Is used for the if statement which is followed by an else statement. It has two conventions, one of which is simply <non-if-statements> the other one it the basic if-else structure.
- **<unmatched_if>:** Is for constructing an if statement without and else. It has two conventions:
    1. <unmatched_if> → if(<logical_expr>) <statements>. Which is the basic, single if definition.
    2. <unmatched_if> →if(<logical_expr>) <matched_if> else <unmatched_if>. This one is for a making the single if statement, nestable.
- **<logical_operator>:** this non-terminal is used to make a group from basic logical operators:” ==, !=, &&, ||, =>” .
- **<arithmetic_operator> :** is the non-terminal which is used to make a group from basic arithmetic operators: “+, -, /, *”
- **<var>:** type definition of a variable. Language allow int and boolean as types.

4. Descriptions of how nontrivial tokens (comments, identifiers, literals, reserved words). Explanations related to various language criteria such as readability, writability, reliability, etc.

Reserved Words

- **if:** This reserved word is used for decision makings in our language.
- **else:** Just like if, else is also a reserved word and is used for decision makings.
- **int:** This reserved word is used for integer typed variable declaration.
- **boolean:** Just like int, boolean is also a reserved word and is used for boolean typed variable declaration.
- **for:** For loop structures in our language is written by this reserved word.
- **while:** This reserved word is used for while and do-while loop structures in our language.
- **do:** This reserved word is used for do-while loop structures in our language.
- **const:** Constant declaration. Variables declared with this keyword must be defined once.

Identifiers: In our programming language, identifiers could be any string which is formed by ISO basic Latin alphabet.
Comments: Comments start with an "#" symbol in our language. There are only line comments. Multi line comments are not supported.

Motivations behind these was to implement a language which has a common syntax with the commonly used programming languages like C descendant languages, so that the users will not have any difficulties with learning and coding with our programming language which relates to writability and readability. Curly braces were used in the language to increase readability. Also, in terms of reliability, programmer must declare type of a variable and return type of the function. This prevents possible wrong interpretations of a variable.