CS315

**PROJECT 2 REPORT**

**GROUP 5**

**SEC 01**

MUHAMMED BERK KOSE 21704277
FATİH SEVBAN UYANIK 21602486
ALEMDAR SALMOOR 21500430

# PLA

THE PROPOSITIONAL LOGIC ANALYZER

# CONTENTS

## BNF of PLA

**<program>** → identifier { <statements> } | identifier { <empty> }

**<statements>** → <statement>; | <statement>; <statements> | comment | comment <statements>

**<statement>** → <if_ stmt> | <for_loop> | <for_each> | <while_loop> | <predicate_declaration> |
<predicate_call> | <var_logic_assignment> | <var_logic_declaration> |
<val_logic_declaration> | <var_string_declaration> | <var_string_assignment> |
<val_string_assignment> | <input_statement> | <print_statements> | <var_list_declaration> |
<var_list_assignment> | <val_list_assignment> | <var_num_declaration> |
<var_num_assignment> | <val_num_declaration> | <list_element_assignment>

**<var_num_declaration>** → var_num $identifier | var_num <var_num_assignment>

**<var_num_assignment>** → $identifier = <operation_numeric>

**<val_num_assignment>** → val_num $identifier = <operation_numeric>

**<var_list_declaration>** → var_list identifier | var_list <var_list_assignment>

**<var_list_assignment>** → identifier = <list>

**<val_list_assignment>** → val_list identifier = <list>

**<var_string_declaration>** → var_string identifier | var_string <var_string_assignment>

**<var_string_assignment>** → identifier = string

**<val_string_assignment>** → val_string identifier = string

**<var_logic_declaration>** → var_logic identifier | var_logic <var_logic_assignment>

**<var_logic_assignment>** → identifier = <operation_boolean>

**<val_logic_declaration>** → val_logic identifier = <operation_boolean>

**<operation_numeric>** → <operation_numeric> <operator_sub_sum> <term_numeric> |
<term_numeric>

**<term_numeric>** → <term_numeric> <operator_mult_div> <factor_numeric> | <factor_numeric>

**<factor_numeric>** → ( <operation_numeric> ) | <property_numeric>

**<operation_boolean>** → <operation_boolean><operator_boolean><term_boolean> |
<term_boolean>

**<term_boolean>** → (<logic_expression> ) | <property_boolean> | ~ <property_boolean>

**<property>** → <operation_numeric> | string | <operation_boolean> | <list>

**<property_numeric>** → integer | double | $identifier | $<list_element_call>

**<property_boolean>** → true | false | <predicate_call> | identifier | <list_element_call>

**<property_list>** → <empty> | <property> | <property>, <property_list>

**<list>** → [<property_list>]

**<list_element_call>** → identifier[<operation_numeric>]

**<list_element_assignment>** → <list_element_call> = <property>

**<operator_boolean>** → & | '|' | '|~|' | <==> | => | ~& | '~|'

**<operator_mult_div>** → * | /

**<operator_sub_sum>** → + | -

**<comparison_operator>** → < | > | <= | >= | == | !=

**<predicate_declaration>** → predicate identifier ( <param_list> ) :: {
<statements_predicate> return <logic_expression> ; }

**<param_list>** → <empty> | <param> | <param>, <param_list>

**<param>** → identifier | $identifier

**<statements_predicate>** → <statements> | <empty>

**<predicate_call>** → identifier ( <property_list> )

**<logic_expression>** → <logic_expression_numeric> | <operation_boolean>

**<logic_expression_numeric>** →
<operation_numeric><comparison_operator><operation_numeric>

**<if_stmt>** → if (<logic_expression>) { <statements> } | if ( <logic_expresssion> ) {<statements>}
else {<statements>}

**<for_loop>** → for (<var_assignment>; <logic_expression>; <var_assignment>) { <statements> }

**<for_each>** → for ( identifier: <list>) { <statements> } | for ( identifier: identifier) { <statements> }

**<while_loop>** → while (<logic_expression>){<statements>}

**<input_statement>** → input<< identifier | input<< $identifier

**<print_statements>** → <print> | <println>

**<print>** → print(<property_list>)

**<println>** → println(<property_list>)

**<empty>** → ε

## Language Constructs

**<program>** The program consists of statements which are written in curly brackets. In addition, the program can also be empty. The name of the program is given before the curly brackets.

**<statements>** Statements are a sequence of comments and statements. After every statement there must be a semicolon sign.

**<statement>** Statement is a useful/meaningful code unit which defines how the program will work. Statements can be if statements, loop statements, predicate statements, various declaration statements, print statements and input statements.

**<var_num_declaration>** Defines how the number type variables should be declared. In number types, there has to be a dollar sign before the identifier which indicates that the variable is a number. For example: var_num $a = 10; or var_num $b or var_num $c = $a;

**<var_num_assignment>** Defines, how the assignment of the number type variables should be done. After the assignment, there has to be a <operation_numeric>. For example: $b = 250; or $num = $temp or $myNumber = 562.226 or $num = 52 + 98;

**<val_num_assignment>** Defines, how the constants of type number should be declared. In number types, there has to be a dollar sign before the identifier which indicates that the constant is a number. For example: val_num $a = 10 + 35;  or val_num $c = $a or val_num $temp = $myList[5];

**<var_list_declaration>** Defines how the list type variables should be declared. For example: var_list myList = [10, 666, “dasdasdsa”, true, 4+5];

**<var_list_assignment>** Defines, how the assignment of the list type variables should be done. For example: tempList= [1423, 642, “string awesome”, false, true];

**<val_list_assignment>** → Defines, how the constants of the list type should be declared. For example: val_list constList= [322, 1337, “this list is constant”, false, true];

**<var_string_declaration>**  Defines, how the assignment of the string type variables should be done. For example: var_string name= “Fatih Sevban”;

**<var_string_assignment>**  Defines, how the assignment of the string type variables should be done. In the assignment, only a string has to be assigned. For example: name= “Fatih Sevban”;

**<val_string_assignment>** Defines, how the constants of string type should be declared. For example: val_string name= “Fatih Sevban”;

**<var_logic_declaration>** Defines how the boolean type variables should be declared. For example: var_logic a = true; or var_logic b; or var_logic c = a | b;

**<var_logic_assignment>**  Defines, how the assignment of the boolean type variables should be done. After the assignment, there has to be a <propert_boolean>. For example: b = true; or num = temp; or myBool = a & c;

**<val_logic_declaration>** Defines, how the constants of type boolean should be declared. For example: val_logic a = true;  or val_logic c = a; or val_logic temp = true & a;

**<operation_numeric>** Defines how the numeric operations should be done. In order to have precedence rules and leftmost recursion, <operation_numeric> is categorised to sub groups such as <operator_sub_sum> and <term_numeric>.

**<term_numeric>** <term_numeric> is an auxiliary tag that enables to make numerical operations. It provides PLA a precedence rule for multiplication and division over subtraction and addition.

**<factor_numeric>** Enables to have precedence rule for parentheses. <operation_numeric>, <term_numeric>, <factor_numeric> works recursively to solve mathematical equations correctly according to the priority of the given elements. <property_numeric> determines the end of the recursion.

**<operation_boolean>** Similar to operation_numeric, defines how the boolean operations should be done. Again, in order to have precedence rules and leftmost recursion, <operation_boolean> is categorised to sub groups such as <operator_boolean> and <term_boolean>.

**<term_boolean>** Enables to have precedence rule for parentheses that contains numerical and propositional logic expressions. . <operation_boolean> and <term_boolean> works recursively to solve logical equations correctly according to the priority rule of the given elements. <property_boolean> determines the end of the recursion.

**<property>** Defines all of the properties such as list, string, <operation_numeric> which contains integers, doubles and mathematical operations and <operation_boolean> which contains true, false and logical operations.

**<property_numeric>** Consists of integers, doubles, identifiers that contains a number indicator($ sign) in front of them and list element calls that have a numerical value. Also, the list elements should be indicated with a number indicator in front of them. For example 5, 6, 7.95, $myNum, $myList[5]

**<property_boolean>** Consists of true, false, identifiers that contains logical truth values, list elements having a value of boolean type and predicate call which returns true or false.

**<property_list>** Defines a list of properties. For example 5, “Ali”, true, 5+7.

**<list>** list is a data type that can be assigned to variables and constants that are entitled to have a list value. Lists in PLA are enclosed in square brackets which include a <property_list>. For example [5, “Ali”, true].

**<list_element_call>** is derived from <property> and is used to access a list element that is in a specific position in the list. In PLA, positions start from 0. For example myList[5] gives the 6th element in the list.

**<list_element_assignment>** is used to assign a value to the list element. As the production rule suggests, <list_element_assignment> derives into <list_element_call> = <property>. For example, myList[3] = 5 assigns numeric value of 5 to the 4th position of myList where myList is a list declared before.

**<operator_mult_div>** produces multiplication (*) or division (/) signs.

**<operator_sub_sum>** produces addition (+) or subtraction (-) signs.

**<comparison operator>** produces comparison operators such as less than, less than or equal to, greater than, greater than or equal to, equality or inequality.

**<operator_boolean>** produces boolean operators. For example: and, or, implication, biconditional…

**<predicate_declaration>** production rule of how predicates are declared in PLA. Production rule tells us that predicate declaration starts with a reserved word *predicate* followed by an identifier. The identifier is followed by a <param_list> in parenthesis. After declaring the <param_list>, "::" is used to indicate that this is a predicate declaration. Finally, there is the predicate body that is enclosed in braces. In the body of the predicate, there are <statements_predicate> return <logic_expression> ;. This makes sure that predicate returns value that is equivalent to a truth value. Example of a predicate declaration:

```
predicate myPredicate(value)::{
        var temp; temp = value;
        return temp;
};
```

**<param_list>** It is a list of parameters that are used in declaring a predicate. For example: logicValue, $number

**<param>** consists of identifiers that are used as indicators of variables, constants and list element calls. $ sign is used to emphasize that the parameter is a number type.

**<statements_predicate>** it may have statements or it can be empty. The purpose of having this tag is to have predicates that have only return statements.

**<predicate_call>** describes how predicates are invoked: identifier that corresponds to a predicate name followed by a <property_list> in parenthesis. The following is an example of an invocation of a predicate which is declared in advance: myPredicate(nValue);

**<logic_expression>** derives either to the result of numeric comparison such as "5 < 6" or <operation_boolean> that eventually derives a result that correspond to a logical truth value. For example: 5+3 < 2+1 or true | c

**<logic_expression_numeric>** always produces a result of comparison between two numeric expressions. For example: 99 <= 100 or 5 +7 > 2+5.

**<if_stmt>** produces two PLA language constructs. It either produces *if* construct or *if/else* construct. In both cases keyword if is followed by a <logic_expression> surrounded by parenthesis and statements block that is to be executed on the basis of the <logic_expression>. statements block is always enclosed by braces regardless of the number of statements. In case of *if/else* statement, *else* keyword follows the closing brace of the first brace of the if clause. For example:

```
 if(5 < 3) {
         ans = “impossible”;
};
```

Or

```
if($yourFace < ($myFace) * 100) {
         ans = “yourFace is smaller”;
} else {
         ans = “yourFace is bigger * 100”;
};
```

**<for_loop>** describes the usual for loop that loops on the defined value and halts when <logic_expression> clause corresponds to logical false.

**<for_each>** describes a similar construct to the usual for loop. However, in *for each* loop, instead of declaring and initializing a loop counter, we iterate over a defined list. Example of a for each loop in PLA:

```
for (looper: myList) {
         println(looper);
};
```

**<while_loop>** describes a familiar while loop that loops on the basis of a logical truth value. For instance:

```
while(i < 50) {
     println("How Are You?");
     i = i + 1;
};
```

**<input_statement>** describes, as the name indicates, an input statement in PLA. Keyword input is followed by “<<” and an identifier name, or and identifier name with a dollar sign front of it to indicate numeric value. When the PLA user provides an input, this value is assigned to the provided identifier. Example for an input statement: input<< sVar; or input<< $nVar;

**<print_statements>** produces two Output statements in PLA.

**<print>** is derived from <print_statements> and describes the output statement after which output cursor stays on the same line. In the print function, multiple parameters can be added through a property list.

**<println>** is derived from <print_statements> and describes the output statement which  moves the output cursor to the next line after its output.Similar to print function, multiple parameters can be added through a property list to the println function,
**<empty>** empty.

## Terminals of PLA

**identifier** - this terminal produces tokens that are used in the named constructs such as predicates, variables, etc.

**comment** - comments of PLA, example comments: */* This is a comment */*, */*This is another comment */* .

**var** - terminal for variable declaration in PLA.

**val** - terminal for constant declaration in PLA.

“**=**” - terminal of assignment operator in PLA.

“**(**“ left and “**)**” right parentheses of PLA.

“**[**“ left and “**]**” right square brackets of PLA.

“**{**“ left and “**}**” right curly brackets of PLA.

“**!**” - Exclamation mark of PLA

“**,**” - Comma in PLA

“ **|** ” - Terminal corresponding to logical OR, also called disjunction

“ **&** ” - Terminal corresponding to logical AND, also called conjunction

“ **~** ” - Terminal corresponding to logical NOT, also called negation

“ **~&** ” - Terminal corresponding to logical NAND, also called alternative denial

“ **~|** ” - Terminal corresponding to logical NOR, also called joint denial

“ **|~|** ” - Terminal corresponding to logical XOR, also called exclusive disjunction

“ **=>** “ - Terminal corresponding to logical IMPLICATION

“ **<==>** “ - Terminal corresponding to logical IF_AND_ONLY_IF, also called biconditional

“ **<** “ - Terminal corresponding to less than

“ **<=** ” - Terminal corresponding to less than or equal to

“ **>** ” - Terminal corresponding to greater than

“ **>=** ” - Terminal corresponding to greater than or equal to

“ **predicate** “ - Keyword used in predicate declaration that precedes predicate name

“ **::** ” - in PLA double colon is used in predicate declaration after parentheses that include <property_list>

“ **:** “ - colon in PLA

“ **print** “ - Terminal used in printing an output without jumping to the next line.

“ **println** “ - Terminal used in printing an output while jumping to the next line.

“ **input<<** “ -  Terminal used to get an input from the user.

“ **==** “ - Terminal corresponding to is equal.

“ **!=** “ - Terminal corresponding to is not equal.

“ **=** “ - Terminal corresponding to assignment.

terminals corresponding to “**+**” addition, “**-**” subtraction, “*****” multiplication, “**/**” division in PLA.

# Nontrivial tokens

## Comments

**Comments** in the PLA are enclosed between “/*” and “*/” and span single line. The motivation is to provide a commenting capability with an already familiar construct that is already used in “C Family Languages” to people that use PLA.

## Identifiers

**Identifiers** in PLA for numeric variables start with a dollar sign ($) followed by an either uppercase or lowercase letter followed by any number of characters. Identifiers for other types start with uppercase or lowercase letter followed by any number of characters.

## Literals

- **String** literals are enclosed between quotation marks.
- **Numeric** literals are represented by Integer and Double literals and are similar as in most of the Programming Languages, i.e *5* is an integer literal and *4.0* is a double literal.
- ***true*** and ***false*** are boolean literals that correspond to logical truth and logical false.

## Reserved words

- **var_num** - precedes identifier in numeric variable declaration.
- **val_num** - precedes identifier in numeric constant declaration.
- **var_logic** - precedes identifier in boolean variable declaration.
- **val_logic** - precedes identifier in boolean constant declaration.
- **var_string** - precedes identifier in string variable declaration.
- **val_string** - precedes identifier in string constant declaration.
- **var_list** - precedes identifier in list variable declaration.
- **val_list** - precedes identifier in list constant declaration.
- **predicate** - is a part of predicate declaration.
- **return** - precedes the return logic expression in the predicate constructs.
- **if** - used as a part of if and if/else statements.
- **else** - used in if/else statements.
- **for** - used in for and foreach constructs.
- **while** - used in the while loop.
- **input** - used as a keyword to the input statement.
- **print** - used in the output statement .
- **println** - used in the output statement that moves the output cursor to the next line.

- **true** - boolean literal corresponding to logical truth.
- **false** - boolean literal corresponding to logical false.

All of those words above can not be used as variable names, constant names and predicate names and are considered as reserved words in PLA.

## Motivation of Design Choices in PLA

### Readability

One of the most important aspects of a programming language being readable is its familiarity to the user. Comments in the language are inserted similar to the most of the C-family Languages that enclose the comment body between “/*” and “*/”. Usage of string, numeric, boolean literals used in PLA, all should be familiar to the people that come with a programming background. All of the constructs include reserved words that try to convey the meaning of the construct. Like ***var_num*** in variable numeric declaration and ***val_logic*** for constant boolean declaration. ***var*** and ***val*** words that are part of the variable and constant declarations were inspired by Kotlin Programming Language which is also emerging in popularity. Input statement has a construct that is easier to read and understand similar to C++ as opposed to Java which declares a Scanner object with System.in parameter and uses the object to read the input. To differentiate numeric identifiers and other identifiers, numeric identifiers are preceded by dollar sign ($). Every if statements and else statements scope is specified with curly braces that eases readability.

### Writability

To improve the Writability, the integers and doubles were combined into one numeric type. PLA has three different constructs for “loops”. The programmer can either choose classical while loop, for loop that declares a variable and iterates on some condition and newer for loop that iterates over a list. The print statements are easier in terms of writability since PLA has print statements for the same line print statement and print statement that moves the cursor to the newline as opposed to C++ where where “endl” should be appended to move the cursor the newline. Also, the print statements are shorter when compared to Java where the print statements should be preceded by System.out. The dollar sign ($) that precedes numeric identifiers also slightly decrease the writability of the language.

### Reliability

Since we have different types for each numerical, logical and string values, the users can rely on which variables they will use, and for the indicator of numeric values. PLA provides a formal kind of writing, which encourages programmers to provide the clearest possible statements. The aim of this is to lower the errors that will occur from PLA to minimum, and make it so that the errors are mostly about syntax. PLA uses $ so that the users can rely on the program. For example, when an identifier has $ sign in front of it, that is to say, it has a numeric value. For numerical and logical operations, the program uses the global standards of precedence rules. In numerical operations the precedence rule comes first for the expressions that are in parenthesis and then, multiplication, division comes first and subtraction, addition comes second. In logical operations, the precedence rule comes first for the expressions that are in parenthesis and then, the precedence rule is followed from left to right.