BILKENT UNIVERSITY
CS-315
PROJECT: NEW PROGRAMMING LANGUAGE 'PINE'
2023-2024 FALL SEMESTER
GROUP 22

Alara Zeybek (22102544) - Section 2
Bora Haliloğlu (22101852) - Section 2
Doğa Elif Konuk (22101920) - Section 2

Contents:

PINE (.pine)

Introduction

Pine is a unique programming language that was created with the goal of improving user-friendliness and code readability in mind. As a result, it has a distinctive syntax. This meticulously crafted syntax is intended to shorten the learning curve for novice programmers while simplifying the development process for experienced developers. Pine allows developers to produce clean, efficient code, making it a good choice for a variety of applications ranging from web development to data analysis.

1.BNF

```
<program> ::= <function_declarations> | <comment_statement>
<function_declarations> ::= <empty_statement> | <function_declaration>|
<function_declaration> <function_declarations>
<function_declaration> ::= <function_declarator> <block> | <function_declarator> ;
<block> ::= {} | {<block_statements>}
<block_statements> ::=  <statements>
<statements> ::= <empty_statement> | <statement> | <statement> <statements>
<statement> ::=  <expression> ; | <conditional_statement> | <input_statement>; |
<output_statement> ; | <wloop_statement> | <roll_statement> | <return_statement>; |
<comment_statement>
<expression> ::= <array_initialization> | <array_declaration> | <variable_declaration> |
<assignment> | <wo_assignment_expression> | <array_index_assign> | <array_index_access>
| <constant_declaration>
<function_declarator> ::= fun <identifier> ( <parameter_list> ) |  fun <identifier> ()
<parameter_list> ::= <parameter_list> , <parameter> | <parameter>
<parameter> ::= <identifier> | <integer>
<variable_declaration> ::= <type> <variable_declarators>
<variable_declarators> ::= <variable_declarator>
<variable_declarator> ::= <identifier> | <assignment>
<assignment> ::= <identifier> <assignment_operator> <math_expression> | <identifier>
<assignment_operator> <input_statement>
<array_declaration> ::= <array_dec_one> | <array_dec_multi>
<array_dec_one> ::= <type> <identifier>  <array_dimension_one>
<array_dec_multi> ::=  <type> <identifier> <array_dimension_multi>
<array_dimension_one> ::= [ <integer> ]
<array_dimension_multi> ::= <array_dimension_one> <array_dimension_one> |
<array_dimension_multi> <array_dimension_one>
<array_initialization> ::= <array_init_one> | <array_init_multi>
<array_init_one> ::= <array_dec_one>  == <array_dimension_init_one> |  <identifier>  ==
<array_dimension_init_one>
<array_init_multi> ::= <array_dec_multi>  == <array_dimension_init_multi> |  <identifier>  ==
<array_dimension_init_multi>

<array_dimension_init_one> ::= { <int_list> }
<array_dimension_init_multi> ::= {<array_dim_init_multi_element> }
```

<array_dim_init_multi_element> ::= <array_dimension_init_one> **,** <array_dimension_init_one> |
<array_dim_init_multi_element> , <array_dimension_init_one>
<int_list> ::= <integer> | <boolean> | <integer>**,**<int_list> | <boolean> **,** <int_list>
<array_index_access> ::= <identifier><array_index_access_one_element> |
<identifier><array_index_access_multi_element>
<array_index_access_one_element> ::=**[** <boolean> **] | [** <integer> **] | [** <identifier> **]**
<array_index_access_multi_element> ::= <array_index_access_one_element>
<array_index_access_one_element>
   | <array_index_access_multi_element> <array_index_access_one_element>
<array_index_assign> ::= <array_index_access> <assignment_operator> <math_expression> |
<array_index_access> <assignment_operator> <input_statement>
<array_length> ::= <integer>
<wo_assignment_expression> ::= <logical_expression> | <function_invocation>
<math_expression> ::= <integer> **$** <term> | <integer> **#** <term> | <term>
<term> ::= <term> ***** <factor> | <term> **/** <factor> | <factor>
<factor> ::= <factor> **^** <base> | <factor> **/&** <base>
<base> ::= <integer> | <identifier> | <wo_assignment_expression> | **(** <math_expression> **)** |
<array_index_access>
<function_invocation> ::= <identifier> **(** <argument_list> **)** | <identifier> **()**
<argument_list> ::= <value> | <argument_list> **,** <value>
<function_name> ::= <identifier>
<conditional_statement> ::= <if_statement>| <else_statement> | <else_if_statements>
<if _statement> ::= **pro (** <logical_expression> **)** <block>
<else_statement> ::= <if _statement> **con** <block> | <else_if_statements> **con** <block>
<else_if_statements> ::= <if _statement> <else_if_statement> | <else_if_statements>
<else_if_statement>
<else_if_statement> ::= **pros** (<logical_expression>) <block>
<logical_expression> ::= <logical_block> | **!** <logical_expression>
<logical_block> ::= <logical_term> | <logical_block> **|** <logical_term>
<logical_term> ::= <logical_factor> | <logical_term> **&** <logical_factor>
<logical_factor>::= <eq_comparison_expression> | <boolean>
<eq_comparison_expression> ::= <value> **=** <value> | <value> **!=** <value> | smaller_than> |
<greater_than>
<smaller_than> ::= <value> **<** <value> | <value> **<=** <value>
<greater_than> ::= <value> **>** <value> | <value> **>=** <value>
<wloop_statement> ::= **wloop (** <logical_expression> **)** <block>
<roll_statement>::= **roll (**<roll_init> **;** <logical_expression> **;** <assignment> **)** <block>
<roll_init> ::= <variable_declaration> | <assignment>
<comment_statement> ::= <single_line_comment> | <multiline_comment>
<single_line_comment> ::= **<>** <string>
<multiline_comment> ::= **<<** <string> **>>**
<input_statement> ::= **pin ( )**
<output_statement> ::= **pout (** <string> **) | pout (** <identifier> **) | pout (** <integer> **) | pout (**
<wo_assignment_expression> **) | pout (** <array_index_access> **)**
<return_statement> ::= **return** <math_expression> **;** | **return ;**
<constant_declaration> ::= **cint** <type> <variable_declarator>
<empty_statement> ::= **;**
<value> ::= <identifier> | <integer> | <literal> | <array_index_access>

<literal> ::= <boolean>
<boolean> ::= <true>| <false>
<true> ::= **1**
<false> ::= **0**
<unsigned_integer> ::= <digit> | <digit> <unsigned_integer>
<integer> ::= <sign> <unsigned_integer> | <unsigned_integer>
<identifier> ::= <letter> | <letter> <identifier> except <keyword>
<string> ::= <char> | <special_char> | <char><string> | <special_char><string>
<char> ::= <letter> | <digit> | <special_char>
<special_char> ::= **' ' | ! | "" | # | $ | % | & | / | \ | == | = | != | > | < | + | - | * | ^ | (|) | { | } | [|] ; | ,**
<letter> ::= **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**
<assignment_operator> ::= **==**
<equ_operator> ::= **=**
<inequ_operator> ::= **!=**
<greater_ex> ::= **>**
<smaller_ex> ::= **<**
<add_op> ::= **$**
<sub_op> ::= **#**
<mult_op> ::= *****
<div_op> ::= **/**
<mod_op> ::= **/&**
<exp_op> ::= **^**
<left_paren> ::= **(**
<right_paren> ::= **)**
<left_bra> ::= **{**
<right_bra> ::= **}**
<left_sqbra> ::= **[**
<right_sqbra> ::= **]**
<semicolon> ::= **;**
<multiline_comment_open> ::= **<<**
<multiline_comment_close> ::= **>>**
<comma> ::= **,**
<exc_mark> ::= **!**
<or_op> ::= **|**
<and_op> ::= **&**
<sign> ::= **+** | **-**
<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<type> ::= **pint**
<keyword> ::= **pint** | **cint** | **pin** | **pout** | **wloop** | **roll** | **fun** | **pro** | **pros** | **con**

Important Note: Terminal symbols returned by the lex implementation are included at the end of the BNF description of the language. Note that instead of using the new definitions, terminal symbols are used by their nature in the upper definitions. This decision aims to increase the readability of the BNF notation of the language.

2.Language Description
Pine is a newly created programming language to ease the usability and readability for the users. Therefore, it has a specially developed syntax.

2.1 Integer Type
Pine is a programming language to especially manipulate integer operations. Therefore, its main variable type is integers and they can be defined by using the '*pint*' keyword. In Pine, there are also some predefined string constants in order to improve readability for some specific user prompts.

2.2 Variable Declaration
To make our users comfortable with our new language, we follow the traditional way of variable declaration. In Pine, you can declare a variable by first writing the type and the name of the variable, then an assignment operator and the value to keep. As assignment operator, pine uses '=='. So the format would seem like the following: pint variable_name == 5;

Note: In pine you cannot declare multiple variables at once:
pint a,b,c == 4;

2.3 Constant Variables Definitions
Pine defines a constant variable in two words, '*cint*' and '*pint*' respectively. By including the '*cint*' keyword in front of *'pint'* type, it is specified that the integer variable is a constant integer. This increases the usability of the language.

2.4 Data Structure for a Collection of Integers

2.4.1 Array Initialization
Pine uses an accustomed approach on array initialization. The arrays can be initialized right after declaration or some lines after initialization. Like the following:

```
pint a[2] == [1, 2];
OR
pint a;
a == [1,2];
```

2.4.2 Array Index Access
The array initialization of Pine is again very usable. When initializing an array index you use the following format:

```
pint a[2];
a[0] == 1;
a[1] == 2;
```

The indexes used for array operations are not necessarily unsigned integers, signed integers also can be used for array indexes. This feature is designed similar to the feature used in Python language.

2.4.3 Multi-Dimensional Arrays

When initializing an array, number of complementary square brackets between the type and identifier indicates the dimension of the array. By adding complementary square brackets, the dimension of the array can be increased, which increases the usability of the language.

```
pint arr[3] == {1,2,3};
pint arr2[3][2] == {{5,7},{6,4},{3,8}};
arr[1] == 3;
arr2[2][0] == 10;
pint x == arr[0];
```

2.5 Operators and Logical Expressions

There is the syntax of the four arithmetic operators, modulo, exponentiation and logical expressions:

2.5.1 Addition Operation

Summation operator is "**$**" and the format is <term> **$** <term>. By using this operator instead of the traditional "**+**" operator, Pine aims to give its users a fun and extraordinary experience. This increases the usability of the language.

2.5.2 Subtraction Operation

Subtraction operator is "**#**" and the format is <term> **#** <term>. The usage of this operator is different than the traditional "**-**" operator, which is a design decision to create an entertaining and innovative experience for the users. This enhances the usability of Pine.

2.5.3 Multiplication Operation

Multiplication operator is "*" and the format is <term> * <term>.

2.5.4 Division Operation

Division operator is "/" and the format is <term> / <term>.

2.5.5 Modulus Operation

Modulo operator is "/&" and the format is <term> /& <term>.

2.5.6 Exponentiation Operation

Exponential operator is "^" and the format is <term> ^ <term>.

2.5.7 AND Expression

The and expression is "&" and the format is <logical_term> & <logical_term>.

2.5.8 OR Expression
The and expression is “|” and the format is <logical_term> | <logical_term>.

2.5.9 Equality and Inequality Expression
The equality expression is “=” and the format is <logical_term> = <logical_term>.
The inequality expression is “!=” and the format is <logical_term> != <logical_term>.

2.5.10 Smaller Than and Greater Than
The smaller than expression is either “<” and “<=” indicating “less than” and “less than or equal to” logical expressions respectively. The format for this expression is <value> < <value> or <value> <= <value>.

The greater than expression is either “>” and “>=” indicating “greater than” and “greater than or equal to” logical expressions respectively. The format for this expression is <value> > <value> or <value> >= <value>.

2.6 Parentheses Usage
The parentheses in Pine are used for taking function parameters “fun foo(a,b,c)”, calling a function (foo(a,b,c)) with the said parameters. Moreover, it can be used for conditional statements and loops (pro, con, pros, roll, wloop). It can also be used in special system functions for input and output (“pin()”, “pout()”).

2.7 Conditional Statements
In Pine, the logic is the same as it is in the traditional if, else if and else statements relations. Only the syntax differs. “pro” corresponds to “if”, “pros” corresponds to “else if” and “con” corresponds to “else” in Java.

2.7.1 If Statement
The if statement is called by the keyword ‘pro’. After the pro keyword, the user can put the boolean expression inside the parentheses, then open brackets and write the code part which will be executed if the statement is correct. Here is an example for the syntax:
```
pro(1>0){
…
}
```

2.7.2 Else If Statement
The else if statement is called by the keyword ‘pros’. Here is an example for the syntax:
```
pro(4>3){
…
}
pros(2<3){
…
}
```

2.7.3 Else Statement

The else if statement is called by the keyword ‘con’. Here is an example for the syntax:

```
pro(4>3){
…
}
pros(2>3){
…
}
//desired number of pros can be listed here similar to “else if” blocks
con{
…
}
```

2.7.4 Others

Other than if, else if and else statements, in Pine there aren’t any other structures to define conditional statements. (i.e. in Java, there is also the switch-case statement.)

2.8 Looping Statements

In Pine, there are two types of looping statements: wloop, roll.

2.8.1 wloop

wloop can be seen as the Pine version of the traditional while loop. The declaration and bracket usage are very similar to the Java version of while loop but in Pine the keyword is wloop. After the keyword, user opens parentheses and puts the expression. Then, the user can write the code inside the brackets that come after the right parenthese and the code will be executed until the expression defined after whoop is not true. Here is an example of whoop syntax:

```
wloop(1){
…
}
```

2.8.2 roll

roll can be seen as the Pine version of the traditional for loop. The declaration is also very similar to the Java version of the for loop but in Pine the keyword is ‘roll’. After the keyword, user opens parentheses and puts the expression. Here is an example of roll syntax:

```
roll(pint i == 5; i < 10; i++){
…
}
```

2.9 Input/Output Statements

In Pine, to get an input from the user, ‘pin’ keyword is used: *pin()* is a function which doesn’t require any parameters and also doesn’t return anything.

To print an output to the user prompt, ‘pout’ keyword is used: Again, *pout()* is a function which doesn’t require any parameters but returns integers or constant strings.

2.10 Function Definitions
The function definitions in Pine start with the “fun” keyword. This keyword is followed with the identifier of the function and the parameters. The functions can take integers and integer arrays as parameters.. Hence the overall look of a simple function is like the following:

```
fun foo(a, b, c){
        return a;
}
```

In Pine, you can pass an integer or an integer array. Following this, after the operations inside the function you can return again an integer or an integer array.

2.11 Commenting
Pine allows users to write single and multiple comments. Inorder to write single comment line users need to follow the following syntax:

```
<> I am a comment
```

To write a multiple line comment, users need to follow the following syntax:

```
<< I am a comment
I love being a comment >>
```

3. Example Programs

3.1 First Program

```
fun main(){
        pout(“Enter 3 values x, y, z (respectively) which are not zero”);
        pout(“Enter x”);
        pint x == pin();
        pout(“Enter y”);
        pint y == pin();
        pout(“Enter z”);
        pint z == pin();
        wloop(x = 0){
                pout(“Please enter a number other than 0”);
                x == pin();
        }
        wloop(y = 0){
                pout(“Please enter a number other than 0”);
                y == pin();
        }
        wloop(z = 0){
                pout(“Please enter a number other than 0”);
                z == pin();
        }
        pout(“x*y*z is equal to ”);
        pint sum == x*y*z;
```

```
            pout(sum);
}
```

3.2 Second Program

```
fun greater(p ,q){
        pout("greater ");
        pout("p: ");
        pout(p);
        pout("q: ");
        pout(q);
        pro(p < q){
            return q;
}
            return p;
}
fun main(){
pint firstArr[4] == {5,0,3,-7};
pint secondArr[3] == {9, -2, -1};
roll(pint a == 0; a < 4;  a == a $ 1){
        roll(pint b == 0; b < 4; b == b $ 1){
                c == greater(firstArr[a], secondArr[b]);
                pout("a, b, c (respectively)");
                pout(firstArr[a]);
                pout(secondArr[b]);
                pout(c);
}
}
}
```

3.3 Third Program

```
cint a == pin();
pint b == pin();
pro(a = b){
            pout("The variables are the same");
}
pros( a > b){
            pint diff == 0;
            pout("The first one is bigger than the second one");
            wloop(a > b){
                        b == b $ 1;
                        diff == diff $  1;
            }
```

```
        pout(“the difference is”);
        pout(diff);
}
```

3.4 Fourth Program

```
fun main(){
        pint a[3];
        a == {9,5,1};
        roll(pint b == 0; b < 3; b == b $ 1){
              pint c == 0;
              pout(a[c]);
        }
}
```

3.5 Fifth Program

```
fun fibonacci(d){
<<
The below function calculates the fibonacci iteratively
>>
        pint c == 0;
        roll(pint b == 0; b < d; b == b $ 1){
                c == c $ b;
        }
        return 0;
}
```

3.6 SixthProgram

```
fun fibonacci(d){
<<
The below function takes the arr2 as a parameter and iterates over the array.
>>
        pint c == 2*3;
        roll(pint b == 0; b <= d; b == b $ 1){
                c == c $ b;
        }
        return c;
}
fun main(){
        <>The below statement is initializing an array of length 4 and a variable a which is equal
        to 0
        pint arr2[5];
        pint a == 0;
        wloop(a < 4){
                arr2[a] == pin();
                a == a $ 1;
        }
        pint b == fibonacci(4);
        arr2[4] == b;
}
```

3.7 Seventh Program

We expect to see an error message! Because in our language if you are using pros you cannot use con before it.

Input Code:

```
fun greater(p ,q){
<< This
is
a
multi
comment !!
>>
    pro(p < q){
        return q;
        pint a == 2;
    }
    con{
        pint a == 4;
        return a;
    }
    pros(1<4){
        return p;
    }
    con{
        return q;
    }
return q;
}
```

The output:

```
FUNCTION IDENTIFIER LP IDENTIFIER IDENTIFIER RP LBRACKET
ML_COMMENT
IF LP IDENTIFIER LT IDENTIFIER RP LBRACKET
RETURN IDENTIFIER SC
INT_TYPE IDENTIFIER ASSIGN_OP CONST SC
RBRACKET
```

```
ELSE LBRACKET
INT_TYPE IDENTIFIER ASSIGN_OP CONST SC
RETURN IDENTIFIER SC
RBRACKET
ELSE_IFsyntax error on line 10
```

3.8 Eighth Program

```
fun main(){
        pint sum == 0;
        pint a[2][4] == {{1,2,3,4}, {5,6,7,8}};
        pint smallest == a[0][0];
           roll(pint b == 0; b < 2; b == b $ 1){
                  roll(pint c == 0; c < 4; c == c $ 1){
                         pro(a[b][c] < smallest){
                                a[b][c] == a[b][c] $ smallest;
                                }
                      sum == (a[b][c] # 4) /& 6;
                      }
           }
pout(smallest);
pout(sum);
}
```