Compiler Project Part 1: Language Specification .PJ

Instructor: Dr. Shah Jamal Alam

Group Members: Talal Zahid -- 01815

Urwa Fatima – 00983

# INTRODUCTION:

The .pj (dotpj) is a general-purpose function based imperative programming language. This language is not for production use. It is best suited to complete on the go calculation tasks and lite input handling. This language is for the people to start learning coding without getting into the complexity of programming language and mainly developed to learn the development and working of a compiler. The structure of the programs in .pj is statements based; the state of the program changes with the execution of the statements i.e the program can run without the main function, which is necessary in c++ language. The program ends by identifying the ‘END’ reserved word.

Program → functions | statements |END

# LANGUAGE FEATURES:

The features of this language includes:

* Function call – with parameters passed by reference and value both. Passing the parameter by reference is used in condition when the user need to give the list as a parameter to the function. Whereas passing the parameter by value is easier and faster for the function to use as it doesn’t need to access the value from the memory table

functions → functions | function

function → identifier (optional\_params): { statements}

optional\_params → params | ε

params → datatype param , datatype param | datatype param

statements → statements | return expression ; | ;

* Conditional Statement

Statements → expression ; | if (expression) { statements } else { statements } | else if (expression) {statements}

expression → numeric literal | string literal | identifier | ( expression) | expression [expression] | expression mathematical\_operators expression | expression relational\_operators expression | unary\_operators expression

* Loops

Statements → for ( expression range expression ) { statements} |

while ( expression ) {statements}

* Nested Loops and Conditionals statements
* Scope of Variables

The variable defined with in a block are limited to that block of code.

Statement → var datatype identifier; | var datatype identifier = expression;

* Arrays

Array → datatyepe[optional\_length] Identifier = [optional\_values];

optional\_values → optional\_values | value

value → str val | int val

# Tokens:

* <keyword,val>
* <identifier, val>
* <operator, val>
* <symbol>
* <int, val>
* <str, val>

# Identifiers:

Identifiers are names provided by the user. These are assigned to variables and functions to uniquely identify them to the compiler. The word ‘identifier’ is used for the language specification in place of the values of variable and function name. The following rules are applied to check for the valid identifiers

* The first character of the identifier must be a letter of the alphabet (upper or lowercase) or an underscore ('\_').
* The rest of the identifier name can consist of letters (upper or lowercase), underscores ('\_') or digits (0-9).
* The name can be assigned using a ‘=’ operator.
* Identifier names are case-sensitive e.g:  myname and myName are not the same.
* Examples of invalid identifier names are 2things, this is spaced out and my-name.
* Identifier should not be a keyword.

identifier → ( \_ | A-Z | a-z)(A-Z,a-z, \_ , 0-9)\*

# Keywords:

Keyword are reserved words for the program, which indicates a specific meaning for the compiler. The following case sensitive keywords are reserved from the user to use as identifiers in the program:

|  |  |
| --- | --- |
| WORDS | USES |
| END | The word identifies the end-of-file of the program |
| def | The word is used for the definition of function |
| for | The word for the loop with a specified range |
| range | The word to define the range for the ‘for’ loop |
| while | The word for the conditional loop |
| TRUE | The word to identify Boolean value |
| FALSE | The word to identify Boolean value |
| if | The word used the start of a conditional statement |
| else | The word used to indicate the otherwise condition of the conditional statement |
| elif | The word to use multiple conditions |
| return | The word to identify the start of return statement of a function |
| NULL | The word that is assigned to a variable that is going to have a string type in the program |
| input | The word is used to identify the statement for the user input |
| var | Identify the variable declaration statement |
| int | Keyword for integer data type |
| str | Keyword for string datatype |
| and | The words used of logical operator ‘and’ that are used in conditional statements and bool data type |
| or | The words used of logical operator ‘ or ’ that are used in conditional statements and bool data type |
| not | The words used of logical operator ‘not ’ that are used in conditional statements and bool data type |

Keywords → END | def | for | range | while | TRUE | FALSE | if | else |else if | return | NULL | input | var |int | str | and |or | not

# Data Types:

|  |  |
| --- | --- |
| TYPE | USES |
| int | For integer of size 2 bytes |
| float | For the single precision float point of size 4 bytes |
| str | For the string of 255 characters |
| double | Doubles the size of the datatype it is used with e.g: var double int x = 3; |

# Operators:

Operator → mathematical\_binary\_operator | mathematical\_binary\_operator | relational\_operator | logical\_operator | assignment\_operator

mathematical\_binary\_operator → + | - | \* | / | // | % | \*\*

mathematical\_unary\_operator → + expression | - expression

assignment\_operator → = | += | -= | \*= | /=

logical\_operator → and | or | not

relational\_operator → < | > | == | != | <= | >=

## Mathematical Operators:

All the mathematical operators are only applicable on the integer data type operands except for the plus operator.

|  |  |  |
| --- | --- | --- |
| Binary Operator | Uses | Example |
| + | It is used for the addition of two numeric data type operands or when both operands are string data type then the strings are concatenated | var int a =1;  var int b = 2;  var int c;  c = a + b |
| - | It is used for the subtraction of two numeric data type operands | var int a =1;  var int b = 2;  var int c;  c = a - b |
| / | It is used for the division of two numeric data type operands giving the answer to be a float data type | var int a =1;  var int b = 2;  var float c;  c = a / b |
| // | It is used for the floor division of two numeric data type operands giving the answer to be an integer data type | var int a =1;  var int b = 2;  var int c;  c = a // b |
| \* | It is used for the multiplication of two numeric data type operands used to | var int a =1, b = 2;  var int c;  c = a \* b |
| % | Modulus operator give the remainder of the two operands whose association is from left to right | var int a =1;  var int b = 2;  var int c;  c = a % b |
| \*\* | It is used to find the exponent of the left number to the power equal to right number | var int a =1;  var int b = 2;  var int c;  c = a \*\* b |
| Unary Operator |  |  |
| + | Unary plus used for single integer data type operand |  |
| - | Unary minus used for single integer data type operand |  |
| Increment operators | Placing of these operators before or after the expression define their precedence in the statement |  |
| ++ | Increment operator (pre and post increment) | ++a or a++ |
| -- | Decrement operator | --a or a-- |
|  |  |  |
| Assignment Operator | These operators have the right to left association |  |
| = | It is used to assign the value to the identifier | Var int id\_name = 34 ; |
| += | It is used to assign the value to the identifier of the addition operator for the integer data type operands | a = a + 5 ; |
| -= | It is used to assign the value to the identifier | a = a - 5 ; |
| \*= | It is used to assign the value to the identifier | a = a \* 5 ; |
| /= | It is used to assign the value to the identifier | a = a / 5 ; |

Relational Operators:

These operators are used in between two expressions and their association is from left to right.

Expression → expression relational\_operator expression

|  |  |
| --- | --- |
| Relational Operator | Uses |
| > Token is <GT> | Greater-than is used on the integer and float datatype to compare if left value is greater than the right value. They return a bool return value e.g: if ( a < b){statement} |
| < Token is <LT> | Less-than is used on the integer and float datatype to compare if left value is lesser than the right value |
| == Token is <EQ> | Equal to is used on the integer and float datatype to compare if left value is equals to the right value |
| != Token is <NEQ> | Not Equal is used on the integer and float datatype to compare if left value is not equal to the right value |
| <= Token is <LEQ> | Less-than Equal is used on the integer and float datatype to compare if left value is greater than the right value |
| >= Token is <GEQ> | Greater-than Equal is used on the integer and float datatype to compare if left value is greater than the right value |

## Logical Operators:

The logical operators are used with bool data types and conditional results

|  |  |
| --- | --- |
| Logical Operator | Uses |
| and | True if both the operands are true |
| or | True if either of the operands is true |
| not | True if operand is false (complements the operand) |

## Operators Precedence

* + Use of PEMDAS. Example: Multiplication has higher precedence than addition, 5 + 2 \* 2 = 9
  + () parenthesis can change the order. Example: (5 + 2) \* 2 = 14
  + The operator precedence are listed in the following table. It is in descending order; upper group has higher precedence than the lower ones.
  + When two operators have the same precedence, associativity will determine the order of operation. Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. All the operators will have left-to-right associativity

|  |  |  |
| --- | --- | --- |
|  | Operator | Precedence level |
| 1 | () | Parentheses |
| 2 | \*\* | Exponent |
| 4 | \*, /, //, % | Multiplication, Division, Floor division, Modulus |
| 5 | +, - | Addition, Subtraction |
| 6 | ==, !=, >, >=, <, <=, | Comparisons, Identity, Membership operators |
| 7 | not | Logical not |
| 8 | and | Logical and |
| 9 | or | Logical or |

# Punctuation:

The following punctuation symbols are tokens with denoting specific meaning for the compiler.

Symbols → ( | ) | , | ; | : | { | } | [ | ]

|  |  |
| --- | --- |
| SYMBOL | USES |
| ( | Used to define precedence in expressions or represent the start of the assignment of the parameter to the function |
| ) | Used to define precedence in expression or represent the end of the assignment of the parameter to the function |
| { | Represent the start of statement block e.g: { statement1; statement2; statement3;} |
| } | Represent the end of statement block |
| [ | Declaration of array or Represent the start of index defining of the array e.g: list\_name[index] = 4; |
| ] | Declaration of array or Represent the ending of index defining of array |
| , | Represent the separation between the parameter definition in the function e.e: def func\_name ( para1, para2): |
| ; | Mark the end of statement e.g: variable\_name = 3 ; |
| : | For the definition of function e.g: def func\_name ( para1, para2): |

# Constant:

There are values that does not change throughout the program. There are three kind of literals; string literals, numeric literal and Boolean

String Literals:

These are the sequence of alphanumeric characters, special characters and underscores enclosed in a double quote or single quotes

Str → “[ A-Z+a-z+0-9+whitespaces+ ]” | ‘[A-Z a-z 0-9 ]’ | [All other characters recognized by python]

Numeric Literals:

These are the sequence of numbers

Numeric\_Literal → digit+ | float+

digit → [0-9]+

float → digits\* . digits

Boolean Literals:

Bool → TRUE | FALSE

# Whitespace:

Whitespace such as tab and spaces are token separators. Whereas, newline is counted to keep the track of the line no of the statement

|  |  |
| --- | --- |
| Whitespace | Uses |
| Space ‘ ’ | Not marked as token but is considered as token separator |
| Newline ‘\n’ | It is added to the line number counter to keep the track of the line of code for the error handling |

# Comments:

Comments are ignored by the compiler.

Single line comment is identified by ‘#’ symbol e.g: #comment

Multiple line comment is identified by a start and end with the symbols ‘#/’ e.g: #/ comment #/

Comment → #[string literals] | #/ [A-Z + a-z + 0-9 + [{|}|(|)|[|]||,|;|=|+|@|-|\*|/|&|||<|>|#|!|~]]\* #/