PFORD Language Reference Manual

Bhagyashree Bohra
Deepti Jindal (group leader)
Sharib Khan
Shringika Porwal

INTRODUCTION

The motivation behind PFORD is to provide an easy to learn, easy to understand language, which can be used by novices to quickly familiarize themselves with the art of programming and logical thinking. PFORD also has another neat feature. The compiler can be instructed to generate equivalent code in any of the popular languages like Java.

LEXICAL CONVENTIONS

Token types include identifiers, keywords, constants, strings, operators and separators, where delimiters are used to separate tokens. A token is a sequence of characters that may be matched to any token.

Whitespace

White space is either a blank, tab or new-line. At least one of these is required to separate tokens, otherwise it is ignored.

Comments

PFORD supports C/C++ style comments. A multi-line comment begins with the characters /* and terminates with */ . A single line comment starts with characters // and continues till the end of line. Examples are as below—

```
/* This is an example of
   multi-line
   comment */

// This is an example of single line comment
```

Identifiers

Identifiers basically consist of program variable names for numbers, strings, arrays of functions. An identifier is considered a sequence of letters, digits or underscores (_) that must begin with a letter and is not a keyword. In PFORD lowercase and uppercase letters are distinguished and not considered the same i.e. 'A' is not the same as 'a'.

Keywords

A keyword is an identifier used as reserved word of the language, which cannot be used for naming identifiers such as variables functions etc. Following is the list of keywords for PFORD:

```
begin           startprogram        endprogram      execute
generate        includelib          const           number
int             Float
OR              AND                 NOT             null
endl            while               loop            run
through         skip_iteration      break_loop
if              then                else
output          input               call
readfile        openfile            writefile       closefile
```

Spatial Delimiters

( ) Parentheses are used around parameter lists
{ } Braces are used in loops or in around functions
[ ] Brackets are used to in array declarations
; Semicolon must be at the end of each statement
, Comma are used to separate parameters in a parameter list.
. Dot are used to indicate parameters of an object

DATA TYPES

PFORD supports multiple data types. The types can be categorized into primitives and non-primitives.

The primitives include numbers, strings, Boolean, and the non-primitive include arrays and objects.

Numbers

Numbers include integers and floating point numbers. PFORD will be able to internally figure out whether a number is either a floating point or integer.

Digit: 0..9
Digits: digit digit*
optional_fraction: . digits |empty
optional_exponent: (E (+|-|empty)digits) |empty
num: digits optional_fraction optional_exponent

Examples:
number a= 1;
number b= 1.2;

Strings

Strings are composed of any combination of characters.

string a= "Compilers";
string b= "abc123";

Boolean

Boolean type is used to store true or false values. False is a value of 0, true is any non zero value

Constants

There are several kinds of constants supported by PFORD.
Number constants :
‘const’ <identifier>=number

String Constant :
‘const’ <identifier>=<string>

Arrays

Arrays are used to store many items of one type. To make programming simpler, the indices will be numbered from 1 and not 0. Users can declare arrays of any of the data types mentioned above and also create array of objects (defined below).

Example: number example_array [length];

Hashes

Users can define hash tables that allow a key to be associated with a value. The syntax for a hash table is:

Variable_name hashtable ={ ‘key1’ = value1, ‘key2’=value2 , …};

Built-in functions like get_keys and get_values returns the keys and the values of the hash.

Objects

PFORD allows users to define their own data types. Object can be declared to consist of multiple primitive data types. However, the objects can’t declare functions in their definition of the object. The declaration of an object is done as follows:

```
object object_name {
        data_type name;
        data_type name1;
};
```

After the object is declared, new instances of the object can be created using other variable declarations:

object_name x,y;

VARIABLES

Declaration

IN PFORD, Variables can be declared as
<Variable_type> <variable_name> [=initial value] ;

Where initialization in the declaration is optional.

Scoping

Variables can be assigned scope as either Global or Local.

Global
If a variable is declared outside any block of code, then it has the Global scope and can be used throughout. For example if a variable is declared outside the main function then the variable is available to all functions.

Local
A variable has a local scope if it is declared inside a code block such as inside a function, in an if-then-else construct or inside a loop. Local variable is accessible and available only inside the block in which it is declared.

OPERATORS

Unary operators

Unary operators '-' and '+' act as a prefix to an operand. The operand should be a number. While '+' operator do not change the value of operand, '-' operator returns the result of operation as, negation of given operand.

For example
+(42.05) returns 45.05
-(82.889) returns -82.889

Other examples of unary operators are +=, -=, *=. When these operators are used the variable will perform the operation to itself.

For example
a +=; //this will add a to itself.

Binary operators

Multiplicative operators:

Multiplicative operators take two operands of type number and return the output of the operation. They are grouped left to right.
* is for multiplication
/ is for division
% indicates percent
mod indicates modulo

Additive operators

Binary operators '+' and '-' indicate addition and subtraction respectively.

'+' can either add two numbers or concatenate a string
'-' subtracts numbers

Power operator

Power operator (‘^’) is used to raise an operand of type number to a certain degree. ‘^’ may be followed by optional ‘+’ or ‘-‘ and mandatory number(integer).

Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

These operations can be applied on operands of type number or regular expressions of type number. The result of this comparison is either 0 (True) or 1(False).

Logical operators

Logical operators ‘OR’, ‘AND’ and ‘NOT’ take regular expressions as operands.

OR operator : indicates the ‘logical or’ of the two regular expressions returns true if any of the regular expression is true.

e.g. expr OR expr

AND operator: indicates the ‘logical and’ of two regular expressions and returns true if both of them are true.

e.g. expr AND expr

NOT operator: is a unary operator. It takes a regular expression as operand and negates it.

NOT (expr)

Assignment Operator(=)

Assignment operator ‘=’ is used to assign a value to an identifier. The value can be of type number or string.

Operator precedence

The precedence of operators in an expression is shown in descending order in the following explanation. The operators included in the same subsection have equal precedence.

Precedence Level	Operator	Explanation
1	[], (), function call, !	Array indices, expressions in parenthesis, function calls, negation
2	+, -	Unary Operators positive or negative number
3	*, /, % and mod	Multiplication , division, percent and modular division
4	+, -	Addition and subtraction
5	> , >= , <, <= ,	Relational operators
6	==, !=	Equality operators
7	AND, OR	logical operators
8	=	Assignment operator

Operations on data types

The grammar for doing operations on numbers will be
num operator num

For string the main operator that will be supported is concatenation which is
strings + strings

The grammar for relational operators
A relationalop B

CONDITIONALS

The conditional if statement is used to alter the execution of the program based on whether the specified condition is valid.

The format of the if statement is as follows:

- if (<expression>) then
  {
  program statements;
  …
  }

If the "condition to be tested" is valid (evaluated as true), then the program statements are executed. Or else they are skipped.

- if (<expression>) then
  {
  program statements1;
  …
  }
  else do {
  program statements2;
  …
  }

In this case, if the "condition to be tested" is valid (evaluated as true), then the program statements1 block is executed. If invalid (evaluated as false), the program statement2 block is evaluated.

LOOPS

Simple loop

This loop allows us to iterate through a loop a specified number of times.

```
loop <number> runs {
    program statements;
    …
}
```

This case will cause the program statements specified in the braces to be executed as many times as specified by num.

Specific loop

This loop allows us to iterate through a loop beginning from a "minimum" value and going to the "maximum" value at a specified interval.

```
loop from <number> to <number> interval <number>
{
    program statements;
    …
}
```

This will cause the program statements in braces to be executed, starting from the minimum (beginning) value to the maximum (final) value at an interval specified.

While loop

The first variation allows the user to iterate through the statements in the braces as long as the specified condition is true.

```
while (<expression>) loop through {
        program statements;
                …
        }
```

The second variation allows the user to iterate through the statements as long as the specified condition is true.

```
loop through {
        Program statements;
} while (<expression>)
```

Special Statements

Skip iteration

This statement, when encountered, in any loop construct as specified above, skips the remaining part of the current iteration

```
skip iteration;
```

Break loop

This statement causes the entire loop to be skipped or "broken out of". All the remaining iterations, including the current one will be skipped.

```
break loop;
```

INPUT/OUTPUT

Output

For printing output to the screen, the following is used:

```
output ("string to be outputted" + <identifier>);
```

Strings in double quotes are printed exactly as written. The string equivalent of the variables is printed. In the case of numbers and strings, the value of the variable is printed, whereas for an array or hash its contents are listed. If the user would like to print the individual array element then the user must specify the element with its indices otherwise the default will print the line n times for an array of size n.

User Input

input ("String to be displayed to user", <identifier>, <identifier>)

Here the user enters information from screen. Accordingly, the user inputs values on the command line, which are read into the variables one by one. If the number of variables specified and inputted by the user does not match, then an error message is displayed. Also if the data types do not match (for e.g.: number given when string expected), an error is displayed.

File handling

Read A File

read_file <filename>;

Reads the file specified by the "filename", it will give an error if the specified file does not exist.

Open A File

open_file (<filename>, new|append);

This option can be used to open an existing file with the "filename" using the "append" option or create a new file with the "filename" as the name using the "new" option. It gives an error if file to be opened does not exist or if a file already exists with the same name as the new file being created.

Write Into A File

write_file (<filename>, <string>|<number>);

This function writes the specified string or number (which is converted to a string) in the end of the file.

Close A File

close_file (<filename>);

Closes the file specified. Gives an error if the file specified does not exist.

EXPRESSIONS

Expressions are the next unit of analysis after identifying the tokens.
Expressions can be defined as:

expr: expr op expr --> (op is an operators like +,-, *, /)
expr: digit
expr: (expr)

STATEMENTS

Statements form one coherent unit of a program

Statements can be defined as:

stmt: identifer=expr
| if expr then stmt
| keyord stmt
| loop <number> runs stmt
| loop from <number> to <number> interval <number> stmt
| while expr loop through stmt
| loop through stmt while expr

FUNCTIONS

Function Definition

FunctionDecl
Function Type FunctionDeclarator Parameter? FunctionBody

FunctionType
The functions can be of return type number, string and nothing. Nothing is simply an easier way of saying void.

FunctionDeclarator
Identifier '(' (Parameter)? ')'

Parameter
Parameter parameter-list

Paramater-list
(',' Parameters | /*nothing*/)*

FunctionBody
‘{‘ (Declarations)? Statement‘}’

User defined functions are declared as follows:
return-type function-name(type1 param1, … , typeN paramN)
{
//function body
return retValue;
}

An example using PFORD
number function_add (number a, number b);
{
	return a+b;
}

When you want to call the function from within the main function the user uses call function.

Using the example function_add from before
Main(){
number a=0;
number b= 0;
call function_add (a,b);
}

Built-in functions

- sort will return a sorted array either by alphabetical order or by number order
  sort (array);
- sum will return the summation of the whole number array
  sum(array);
- min will return the minimum of the number array
  x=min(array);
- max will return the max of the array
  x= max(array);
- mean will return the average of the number array
  x= mean(array);
- std_dev will return the standard deviation of a number array
  x=std_dev(array);
- Split will split a string using the delimiter and store the tokens an array.
  Split (string, delimiter)
- Join will join the elements in an array using the ‘separator’ between the elements
  Join (array, “separator”)

- contains this will search for the string_match in the ‘string_tar’ (the target string). Options can be specified. For example, I – would stand for match ignoring case, g for global matches.
  contains (string_tar, string_match, options)
- replace will replace the pattern with replacement in the string word. Options can be I, for ignoring case and g for global which means all instances of the pattern will be replaced.
  replace (word, pattern, replacement, options)
- substring
- to_capital will return a string array or a string in all caps
  to_capital(stringTemp)
- to_lower will return a string array or a string in all lower case
  to_lower(stringTemp)
- get_token
- compare(array, array1)
- complement(array, array1)