Petros: A Multi-purpose Text File Manipulation Language

Language Reference Manual

Joseph Sherrick
js2778@columbia.edu

June 20, 2008

Table of Contents

1 Introduction

Petros is a multi-purpose text-processing programming language. *Petros* utilizes a pattern matching scheme to extract relevant information from a text file. It is designed to provide a flexible framework from which a user can manipulate data by specifying a set of expressions and statements consisting of rules, conditions, and operations. Relevant information pertaining to a specified relationship or location is educed from a text file to provide information or knowledge about the file's content.

2 Lexical Structure

Programs are written using the Unicode character set. The Unicode characters resulting from the lexical translations are reduced to a sequence of elements (Section 2.1) consisting of white space (Section 2.2), comments (Section 2.3), and tokens. Tokens are comprised of identifiers (Section 2.4), keywords (Section 2.5), literals (Section 4), and operators (Section 2.7) of the syntactic grammar.

2.1 Language Elements

Language elements are the various pieces of a *Petros* program. These elements provide the building blocks of text file manipulation by combining elements in a meaningful sequence. Elements consist of tokens, comments, and white space. Tokens are comprised of identifiers, keywords, literals, separators, and operators. White space and comments serve to separate tokens, provide structural organization, and improve syntactic readability. For example, the keywords *else* and *if* can form the conditional statement token *elseif* only if there is no intervening white space or comment.

2.2 White Space

White space represents any non-printable character such as the spacebar, tab, new line, and form feed. White space is not used to dictate the scope of variables or program flow. It instead provides a means to separate tokens and is otherwise ignored by the compiler.

2.3 Comments

The *Petros* programming language provides two kinds of comments:

/* text */	all the text between /* and */ is ignored
text	all the text from // to the end of the line is ignored

Both comment types employ the following properties:

- comments do not nest
- both /* and */ have no special meaning in comments that begin with //
- sequential characters // have no special meaning within comments that begin /* and end with */

As a result, the following example is a single complete comment:

/* this comment /* // /** ends at the end of this sentence */

2.4 Identifiers

Identifiers refer to user-defined variable names which are essential for symbolic processing. An identifier is a sequence of letters and digits, the first character is required to be a letter. An identifier cannot have the same spelling as a keyword or literal. Letters and digits may be draw from the entire character set. This includes all uppercase and lowercase ASCII letters A through Z and a through z as well as ASCII digits 0 through 9. Two identifiers are considered equivalent if they contain the same character for each letter or digit. All identifiers are case sensitive.

2.5 Keywords

Keywords represent a specific meaning to the *Petros* language and cannot be used as identifiers. The comprehensive list of keywords includes:

if elseif else for do while and or delimit print write read file insert remove contains done

2.6 Separators

The following characters are used as separators:

( )	encapsulates the arguments of a statement
{ }	defines the body of a *Petros* statement
[ ]	specifies the structural location of data
;	signifies the end of a statement
,	separates the arguments of a statement

2.7 Operators

2.7.1 Assignment Operator

=	used to initialize a particular *identifier*.

2.7.2 Concatenation Operator

^ used to connect multiple series of characters together. A series of numbers is assumed to be a string.

2.7.3 End of Set Operator

~ used to signify the final value of either rows or columns in the location specification (Section 3.1).

2.7.4 Arithmetic Operators

+	additive operator
-	subtraction operator
*	multiplication operator
/	division operator

2.7.5 Equality and Relational Operators

Equality and relational operators evaluate a particular relation between two entities. These operators return true or false depending on whether the conditional relationship between the two operands holds or not. The equality and relational operators include:

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

2.7.6 Logical Operators

The condition is evaluated *true* or *false* as a Boolean expression. On the basis of the evaluation, the expression invokes some particular action. The conditional operators include:

- *and*: evaluates to *true* if all conditions return *true*; *false* otherwise.
- *or*: evaluates to *true* if at least one condition returns *true*; *false* otherwise.
- *contains*: evaluates to *true* if all or some portion of an entity contains the specified element; *false* otherwise.

2.7.7 Precedence

Expressions are evaluated as left-associative. Parentheses may be used to force precedence. Table 1 depicts the precedence of operators in the *Petros* language from highest to lowest.

Precedence	Operator	Description
1	( )	Parentheses
2	* /	Multiplication and Division
3	+ -	Addition and Subtraction
4	==, !=, >, >=, <, <=	Relational and Equality Operators
5	and or contains	Logical And, Or, Contains

Table 1. Precedence of *Petros* operators. Shown from highest to lowest.

3 Underlying Functions

3.1 Location Specification

[x:y,j:k]

An arbitrary location within a text file is specified in terms of a row and column. The location can be a single row, a range of rows and columns, or a specific column within a row. The range of rows and columns are separated by a colon. Rows and columns are separated by a comma with the first parameter being a row and the second a column. Both row and column numbers begin at the numerical value of one. If a column value is not explicitly stated, it is assumed that the entire row is selected.

3.2 delimit

delimit('symbol', ... 'symbol');

The delimit statement provides the organizational structure for a specified text file. Each delimitation symbol is specified between single quotation marks and separated using commas. A file's delimiting factors can be changed at any point in a program. Expressions and statements apply to the most recently delimited factors.

3.3 print

print(variable);
print([x:y,j:k]);

The print statement displays specified or manipulated data in a text file. Data may be the contents of a particular variable or from a specific location in the file.

3.4 write

write(variable);
write("string");
write([x:y,j:k]);

The write statement specifies arbitrary text to be written to a file. Text may be the contents of a variable, a specified string, or some or all of the contents of another text file. Text is written to a file in sequential order as defined by each write statement.

3.5 read

read(directory, filename);

The read command specifies the location and filename of the input text file. The file location is the first parameter passed to the read statement. The location is stated by specifying a pathname in either a windows or linux operating environment. The filename is the second parameter passed to the read statement. Parameters are separated by a comma.

3.6 file

file(directory, filename);

The file statement specifies the location and filename to perform write operations to. The file location is the first parameter passed to the file statement. The location is stated by specifying a pathname in either a window or linux operating environment. The filename is the second parameter passed to the file statement. Parameters are separated by a comma.

3.7 insert

insert(variable, [x:y,j:k]);
insert("string", [x:y,j:k]);

The insert statement places text in a user specified location of a text file. Text may be the contents of a variable or a specified string. The text to insert is the first argument passed to the insert statement. The location in the file is specified by the second argument passed to the insert statement. Arguments are separated by a comma. The file that is modified by the insert statement is specified by the file statement (Section 3.5).

3.8 remove

remove(variable);
remove("string", [x:y,j:k]);
remove([x:y,j:k]);

The remove statement deletes text in a user specified location of a text file. Text may be the contents of a variable, a specified string, or all the text contained in a particular location. The text to remove is the first argument passed to the remove statement. The location in the file is specified by the second argument passed to the remove statement. Arguments are separated by a comma. If all the text contained in a specified location is desired for deletion, the location is the only argument passed to the remove statement. The file that is modified by the remove statement is specified by the file statement (Section 3.5).

3.9 done

done

Signifies the end of a program.

4 Data Types

The *Petros* language supports the use of string, integer, and floating point data types. Depending on the context of a statement, the value of a statement evaluates to either a number or string. Variable representation is not explicitly defined by the user. *Petros* distinguishes the contents of a variable and evaluates an expression accordingly. The following data types are implicitly defined by the user:

- *integer*: specified by a sequence of digits that do not contain a decimal point.
- *float*: specified by a sequence of digits that contain a single decimal point either beginning, within, or at the end of the sequence.
- *string*: specified within double quotation marks by a sequence of letters, digits, symbols, or white space.

Strings are prohibited from arithmetic evaluations.

5 Statements

The sequence of program execution is controlled by statements. Statements employ user specified logic to make execution based decisions depending on whether a set of conditions are satisified. Conditional consummation may be required before expressional execution or after the subsequent execution of expressions.

5.1 The if and elseif Statement

The if and elseif statement allows conditional execution of a statement or a conditional choice of two or more statements, executing one or the other but not both. The elseif statement is only valid if there exists a preceding if statement; however, an if statement does not require an elseif. Execution continues by making a choice based on the resulting value:

- If the *if* expression is evaluated as *true*, then the enclosed statement(s) are executed.
- If the *if* expression is evaluated as *false*, then the *elseif* expression is evaluated if one exists; otherwise, do nothing.
- If the *elseif* expression is evaluated as *true*, then the enclosed statement(s) are executed.
- If the *elseif* expression is evaluated as *false*, no further action is taken.

5.2 The if-else Statement

The if-else statement allows conditional execution of a statement or a conditional choice of two, executing one or the other but not both. The else statement is only valid if there exists a preceding if statement; however, an if statement does not require an else. Execution continues by making a choice based on the resulting value:

- If the *if* expression is evaluated as *true*, then the enclosed statement(s) are executed.
- If the *if* expression is evaluated as *false*, then the *else* expression is executed if one exists; otherwise, do nothing.

5.3 The while Statement

The while statement executes an expression and a statement repeatedly until the value of the expression is *false*. A while statement is executed by first evaluating the expression. Execution continues by making a choice based on the resulting value:

- If the value is *true*, then the enclosed statement(s) are executed. After statement(s) finish execution, the process reiterates.
- If the value is *false*, then no further action is taken.

5.4 The do Statement

The do statement executes a statement and an expression repeatedly until the value of the expression is *false*. A do statement is executed by first executing the statement. Execution continues by making a choice based on the resulting expression value:

– If the value is *true*, then the entire do statement is executed again.
– If the value is *false*, no further action is taken.

The execution of a do statement always executes the enclosed statement(s) at least once.

5.5 The for Statement

The for statement executes some initialization code, an expression, a statement, and some update code repeatedly until the value of the expression is *false*. The for statement provides the user a capability to run a segment of code by some arbitrary number of iterations. Execution continues by making a choice based on the resulting expression value:

– If the value is *true*, then the entire for statement is executed.
– If the value is *false*, no further action is taken.

6 Expressions

Expressions represent the combination of conditions, values, variables, and operators. They are used to make conditional decisions and arithmetic manipulation as well as the extraction of data based on equity or location.

6.1 Arithmetic Expressions

Arithmetic expressions are invoked by specifying addition (+), subtraction (−), multiplication (*), or division (/) operators. Results are either stored in a variable using the assignment operator or used to evaluate some arbitrary condition. An example arithmetic expression is:

var1 = (25 + 50) * var2 - 10;

Where var2 equals 2, the expression evaluates to var1 equals 140.

6.2 Conditional Expressions

Conditional expressions are invoked by specifying equal to (==), not equal to (!=), greater than (>), greater than or equal to (>=), less than (<), and less than or equal to (<=) operators. Results are used to make decisions about whether arbitrary code segments should be executed as well as matching patterns for data manipulation. Conditional expressions are commonly used for if, elseif, for, and while statements. An example conditional expression is:

```
if(var1 > 100) {
    statement
    ...
}
```

Where *statement* is executed if var1 is greater than 100.

6.3 Logical Expressions

Logical expressions are a sequence of one or more conditional expressions that employ user specified logic to make execution based decisions depending on whether a set of conditions are satifisfied. An example logical expression is:

```
if(var1 == 50 or var2 > 100) {
    statement
    ...
}
```

Where *statement* is executed if either var1 is equal to 50 or var2 is greater than 100.

7 Example Program

In this section, we delineate an example program. The program utilizes both an integer and string data type. The column structure is specified by the *delimit* statement. The *read* keyword identifies both the file location and filename to be read and the *file* keyword identifies the location and filename to write to. The *for* statement iterates four times before ceasing execution. Rows one through four are read and tested to see if they contain the string comprised by the identifier, *var*. If *var* is contained, the numeric value contained in row one through four, column two is stored in the variable *input*, a value of six is added to it and *i*, the string *input=*, and *input* are written to a file. If they do not contain *var*, the *elseif* condition is tested to see whether rows one through four contain a string. If evaluated *true*, the value of *input* is inserted at row *i* and column two. If neither the *if* or *elseif* statements are satisified, the *else* statment is executed where row *i* is deleted.

In the second half of the program, the delimit statement is used to change the column structure. The *for* statement iterates 8 times before ceasing execution. The value of row *i* and column 3 is read and tested against the condition stated by the *while* expression where *while* is executed if *input* is greater than 100 and less than 200. If evaluated *true*, a value of two is added to *input* and stored in *input*. Also, the current value of *i* is concatenated with *input* and printed to the screen. The example program syntax includes:

```
var = "example string";
delimit('!', '?', '@', '&');
read(/home/js2778/, some_file);
file(/home/js2778/, test);
for(i=1; i<5; i++) {
    if([i] contains var) {
        input = [i,2];
        input = input + 6;
        write([i]^ "input="^ input);
    }
    elseif([i] contains "another string") {
        insert(input, [i,2]);
    }
    else {
        remove([i]);
    }
}
delimit('$', '(', ')');
for(i=1; i<9; i++) {
    input = [i,3];
    while(input > 100 and input < 200) {
        input = input + 2;
        print(i^ input);
    }
}
end
```