Petros: A Multi-purpose Text File Manipulation Language

Joseph Sherrick

Computer Science Department, Columbia University
500 West 120th Street, New York, NY 10027
js2778@columbia.edu

1 Introduction

Text file parsing and manipulation is a capability that is necessitated among a broad span of disciplines. It is commonly used for Computer Science related research where empirical results are often deposited with distinct formats in text files so that they can later be deciphered for analysis. Programming languages such as *awk* and *Perl* were originally designed to provide text processing facilities. While *awk* has generally remained unchanged, Perl has been expanded to include a wide range of tasks including system administration, web development, network programming, GUI development, and more.

In this paper we propose *Petros*, a multi-purpose programming language designed for processing text-based data in files. *Petros* utilizes a pattern matching scheme to enable the extraction of relevant information from a text file. The language provides an inherently flexible framework where a user can manipulate data by specifying a set of expressions consisting of rules, conditions, and operations. Resultant information can be used to provide knowledge about the text file's content. Ease of use is critical to facilitate desired results; therefore, our language has been designed to accommodate intuitive syntax and semantics.

We describe the functionality of *Petros* in Section 2. In Section 3 we present the language functionality in greater detail and delineate each of its data types, operators, control structures, and conditional logic. Finally, Section 4 presents a program example to illustrate the syntax and semantics of *Petros*.

2 Language Functionality

Petros is designed to enable text file manipulation in numerous ways to yield desired information or knowledge about a file's content. The generality of *Petros* affords the utility for performing simple or complex text-processing tasks. Relevant information is educed based on equality or location. A user specifies whether to display data that meets a set of conditions, perform arithmetic operations to acquire statistical information, or conduct manipulation techniques such as text insertion and deletion.

Petros provides an environment were data can be structurally organized to incur a set of rows and columns. These structural parameters allow a user to

categorize data and display or perform operations accordingly. Equality and relational operators facilitate the selection of data as well as a user's ability to build complex functions. Conditional operators provide a means for data to be selected if a group or subset of conditions is met. Finally, control structures employ decision making and looping schemes that enable a program to conditionally execute particular statements and operations.

3 Language Attributes

3.1 Identifiers and Keywords

Identifiers are lexical tokens that name entities. Naming entities makes it possible to reference them, which is essential for symbolic processing. *Keywords* represent a specific meaning to the *Petros* language and cannot be used as *identifers*. The comprehensive list of *keywords* includes: *column*, *row*, *int*, *float*, *string*, *if*, *elseif*, *else*, *for*, *do*, *while*, *and*, *or*, *delimit*, *print*, *write*, *readFromFile*, *writeToFile*, *insert*, *remove*, *beginsWith*, *endsWith*, *contains*, *end*.

3.2 Data Types

Data types are the building blocks of any programming language. They define a set of values and the allowable operations on those values. The data types encompassed by the *Petros* language includes:

- *integer*: specified using the *keyword int*. *Integers* represent whole numbers that can be used for relational and conditional operators, loop iterations, and arithmetic operations.
- *float*: specified using the *keyword float*. Floating point numbers represent fractional numbers that can be used for relational and conditional operators as well as arithmetic operations.
- *string*: specified using the *keyword string*. *Strings* represent explicit characters that can be used for pattern matching, equality and conditional operators.

3.3 Simple Assignment Operator

= used to initialize a particular *identifier*.

3.4 Arithmetic Operators

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

3.5 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

3.6 Conditional 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.

3.7 Control Structures

Control structures employ user specified logic to make execution based decisions depending on whether a set of conditions are satisfied. Conditional consummation may be required before expressional execution or after the subsequent execution of expressions. The *Petros* language includes the following control structures:

- *if-then-else*: enables execution of a particular section of code *only if* a particular test evaluates as *true*.
- *while and do-while*: continually executes a block of statements while a particular condition is *true*.
- *for*: repeatedly executes a block of statements until a particular condition is satisified.

4 Example Program

In this section, we delineate an example program. The program defines both an integer and string data type. The column structure is specified by the *delimit* statement. The *readFromFile* keyword identifies both the file location and filename to be read. The *for* control structure iterates five times before ceasing

execution. Rows one through five are read and tested to see if they contain the string comprised by the identifier, *var*. The numeric value contained in row one through five, column two is stored in the variable *input*.

In the second half of the program, the delimit statement is used to change the column structure. The *while* control structure evaluates rows 3 through 27, column 5 while the result is equal to the value stored in *input* or the numerical value of 26. If the *while* control structure evaluates to *true*, column five is displayed to the screen as well as being written to the location and filename specified by the keyword, *writeToFile* and *write*. The example program syntax:

```
string var = 'example string';
int input;
delimit('!', '?', '@', '&');
readFromFile(/home/js2778/, some_file);
for(i=0; i<5; i++) {
    if(row[i] contains var) {
        input = row[i]:column[2];
        input = input + 6;
    }
}
delimit('$', '(', ')');
while(row[3:27]:column[5] == input or 26) {
    print(column[5]);
    writeToFile(/home/js2778/, test);
    write(column[5]);
}
end
```