Petros: A Multi-purpose Text File Manipulation Language

Joseph Sherrick

js2778@columbia.edu

August 11, 2008

Table of Contents

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 present *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 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 reading and writing. 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.

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.

We provide a language tutorial in Section 2. In Section 3 we present the *Petros* language reference manual which specifies all the language intricacies. Section 4 outlines the plan and methodology used for designing and building the *Petros* language. The architectural design is shown in Section 5. Finally, the functionality and correctness test plan is presented in Section 6 as well as the project lessons learned in Section 7.

2 Language Tutorial

The *Petros* language supports 7-bit ASCII characters. Programs are interpreted by invoking the command line argument:

```
java Petros program.pet
```

where all programs must contain the file extension '.pet'. Data types *String*, *int*, and *File* are provided as well as the ability to use arrays excluding *File* arrays. A string is an arbitrary sequence of letters, numbers, or punctuation.

An integer is a sequence of numbers and a file data type specifies an operating system pathname and filename. All program variables are globally scoped and must be defined at the beginning of a program. For example, any program intending to use *identifiers* to reference stored strings, integers, or files would begin a program as:

```
String string_identifier;
StringArray sArray_identifier;
int integer_identifier;
intArray iArray_identifier;
File file_identifier;
```

Once an *identifier* has been defined it can be initialized and set at any point throughout the program. Array sizes are not required to be explicitly defined; however, they must grow in sequential order. For example, consider an array named *a* that currently has five elements beginning at index zero as:

a[0], a[1], ..., a[4];

where a[4] is the fifth and final element in the array. At this point, the next index initialized for *a* must be a[5] and cannot be referenced as a[6] or greater.

Whitespace has no particular meaning and is ignored in a *Petros* program. All statements and expressions are terminated with a semicolon ';' except control statements such as if/else and while. An example while statement is defined as:

```
while (expression) {
    statement(s);
}
```

where *statement(s)* represents one or more statements.

The *Petros* language is predicated with the concept of text file manipulation and therefore, provides functions to read and print from a file as well as writing to a file. Reading and printing from a file can be specified in terms of a location broken into rows and columns. However, columns are not required to be specified. Columns are defined based on delimiters. For example, if we consider a file *identifier* named *file* and initialized as:

file = "/pathname/filename.txt";

we can set delimiting factors by:

file.delimit("*", ".", "?");

where * . and ? are used to separate columns. Arrays are particularly useful for reading from a file. An array can be utilized to read and store multiple row and column data from a file using the *becomes* keyword. For example, rows 3 through 5 and columns 7 through 9 can be read and stored in a *StringArray identifier* named *sA* by:

```
sA becomes file.read(3:7|5:9);
```

In this given example, *sA* would contain nine elements where each string element is stored and indexed in sequential order. Location definitions are not required to be in the above form. A read statement may contain one row as: (row), a range of rows: (row1|row2), a single column in a row: (row:column), a single row with a range of columns: (row1:column1|row1:column2), and a range of rows and columns: (row1:column1|row2:column2). It should also be noted that the end of a row or column may be referenced as: (row1:column1| ∼:∼).

The *Petros* language also provides functionality for arithmetic operations such as addition, subtraction, multiplication, and division. Relational operators such as equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to. Also, logical operators such as *and*, *or*, and *contains* are available. Some examples where *x* is of type *int* are:

```
x = 10 + 10 - (5 * 2) / 5;

x > 5 and x < 20 or x == 25;
```

3 Language Reference Manual

3.1 Lexical Structure

Petros accepts programs encoded using 7-bit ASCII. The Unicode characters resulting from the lexical translations are reduced to a sequence of elements (Section 3.1.1) consisting of white space (Section 3.1.2), comments (Section 3.1.3), and tokens. Tokens are comprised of identifiers (Section 3.1.5), keywords (Section 3.1.7), literals (Section 3.1.4), and operators (Section 3.1.9) of the syntactic grammar.

3.1.1 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.

3.1.2 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.

3.1.3 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 */

3.1.4 Data Types

3.1.4.1 String

Strings are represented by one or a sequence of ASCII characters. Strings are defined by placing double quotation marks around the desired string. If a double quotation mark is desired within the string, it is enabled by specifying two consecutive " characters as:

This "string" here

could be defined as

"This ""string"" here"

3.1.4.2 StringArray

StringArray is a collection of strings. The size of an array does not need to be specified before use. The array simply grows as strings are added to it. Items must be added to an array index in sequential order. Indices begin at zero. A StringArray is defined as:

StringArray *identifier*;

a StringArray is set by:

identifier[*NUM*] = *NUM*; or *identifier*[*identifier*] = *identifier*;
or *identifier*[*intArray*] = *identifier*[*intArray*];

where the *identifier* in brackets can be either an *int* or *intArray*. Although not explicitly shown, the right side of the assignment operator can be with an *identifier* or *NUM* for either case.

3.1.4.3 int

Integers are represented by a sequence of digits.

3.1.4.4 intArray

intArray is a collection of integers. The specification rules for *StringArray* defined in Section 3.1.4.2 hold true for intArray.

3.1.4.5 File

The File data type is used to specify a location and filename within an operating system. The pathname and filename are defined together within double quotation marks. For example, a linux file would be specified as:

File *identifier*;
identifier = "/pathname/filename.txt";

3.1.4.6 true

true is a boolean type. It can be explicitly stated to set the trim function of a File type or the execution of a control statement.

3.1.4.7 false

false is a boolean type. It can be explicitly state to set the trim function of a File type or avoid the execution of a control statement.

3.1.5 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.

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.

3.1.6 Scope

All *identifiers* have global scope. All *identifiers* are declared at the beginning of a program and can be initialized or set after all *identifier* declaration.

Type $identifier_1$;
Type $identifier_2$;
.
.
.
Type $identifier_n$;

where n is the total number of *identifiers*.

3.1.7 Keywords

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

if *else* *while* *becomes* *and* *or* *contains* *delimit* *trim* *print* *printF* *read* *write* *int* *intArray* *String* *StringArray* *File* *true* *false* *newline* *tab*

3.1.8 Separators

The following characters are used as separators:

( )	encapsulates the arguments of an expression
{ }	defines the body of a *Petros* control structure
[ ]	specifies an array index for both *intArray* and *StringArray*
;	signifies the end of a statement
,	separates the arguments of a delimit statement
:	separates row and column locations
|	separates beginning row and column locations from ending row and column locations

3.1.9 Operators

3.1.9.1 Assignment Operator

identifier = *expression*;

The assignment operator evaluates an expression and then binds its value to the variable specified by *identifier*. All *identifiers* must be defined at the beginning of the program and previous to any assignment statement. Data types must be consistent on both sides of the operator. *StringArray* types are a collection of type *String* and are therefore, equivalent types. The same holds true for *intArray* and *int*.

3.1.9.2 becomes Operator

identifier becomes *statement*

The becomes operator evaluates a statement and then binds its value to the variable specified by *identifier*. Becomes is used to store data from a text file. The read statement (Section 3.2.6) is used to specify a location from where to retrieve data to store. Both *String* and *int* data types are available as well as the use of arrays.

3.1.9.3 Concatenation Operator

expression ˆ *expression*

Concatenation is used to connect multiple strings and/or integers. If an *int* type is specified, it is treated as a string of numbers.

3.1.9.4 End of Set Operator

(*NUM*:*NUM*| ∼:∼)

The end of set operator is used to signify the final value of either a row or column in the location specification (Section 3.2.1). In other words, ∼ represents the final row in a file or the final column in a row.

3.1.9.5 Arithmetic Operators

Additive: *expression* + *expression*
Subtraction: *expression* - *expression*
Multiplication: *expression* * *expression*
Division: *expression* / *expression*

Strings are prohibited from arithmetic evaluations.

3.1.9.6 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. With the exception of the *equal to* operator (==), all operators evaluate data types of *int*. The equal to operator can be used to determine whether two strings are equivalent. The equality and relational operators include:

Equal to: *expression* == *expression*
Not equal to: *expression* != *expression*
Greater than: *expression* > *expression*
Greater than or equal to: *expression* >= *expression*
Less than: *expression* < *expression*
Less than or equal to: *expression* <= *expression*

3.1.9.7 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:
expression and *expression*
Evaluates to *true* if all conditions return *true*; *false* otherwise.

or:
expression or *expression*
Evaluates to *true* if at least one condition returns *true*; *false* otherwise. All terms are evaluated regardless of previous result.

contains:
expression contains *expression*
Evaluates to *true* if all or some portion of an entity contains the specified element; *false* otherwise. All data types and arrays are eligible for evaluation. For example, a *string* can be tested to see if it contains an *int*.

3.1.9.8 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.

3.2 Underlying Functions

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

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

3.2.1 Location Specification

An arbitrary location within a text file is specified in terms of a row r and column c where r and c are both integer values or the end of file operator $\sim$ as defined in Section 3.1.9.4. 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 bar, '|'. Rows and columns are separated by a colon with the first parameter being a row and the second a column. Both row and column numbers begin at the numerical value of one. Location arguments are evaluated inclusively. If a column value is not explicitly stated, the entire row is selected. Text file locations are used for both read (Section 3.2.6) and printF (Section 3.2.4) statements.

A single row within a text file is specified as:
(r)

A single column within a row is specified as:
$(r$:c)

A range of rows are defined as:
$(r|r)$ or $(r|\sim)$

A range of rows and columns are defined as:
$(r{:}c|r{:}c)$ or $(r{:}c|\sim{:}\sim)$ or $(r{:}c|\sim{:}c)$ or $(r{:}c|r{:}\sim)$
where the beginning row and column is specified to the left of | and the ending row and column to the right.

3.2.2 delimit

delimit provides the organizational structure for a specified text file. Each delimitation character is specified between double 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. No delimiting factors are set by default and therefore, a row within a text file is not broken into columns. Once a file's delimiters have been set, the default can be restored by specifying a blank character as, "". Delimiters are not returned as part of any column.

delimit("$character_1$", "$character_2$", ..., "$character_n$")

where n is the total number of *characters* and *character* is a single 7-bit ASCII character or space. All delimiters can be unset and returned to default by:

delimit("")

3.2.3 print

print is used to print a user defined string or the contents of an *identifier* to the screen. *Identifiers* of type *int* are treated as a string of digits and are permitted for use within a print statement. Arrays may be printed on an index by index basis. The concatenation operator (Section 3.1.9.3) may be used within a print statement to connect both strings and integers.

print(*expression*)

3.2.4 printF

printF is used to print text from a text file. Desired text is identified by location with the location statement (Section 3.2.1). Text that resides within the beginning and ending column bounds return the delimiters that divide each column.

printF(*location*)

3.2.5 write

write specifies arbitrary text to be written to a file. Text may be the contents of an *identifier* or a specified string. All data types may be written to a text file. Arrays may be printed on an index by index basis. The concatenation operator (Section 3.1.9.3) may be used within a write statement to connect both strings and integers. If the text file being written to does not exist, the write statement creates the file and writes to it sequentially. If the text file does exist, the write statement writes text to the end of the file.

write(*expression*)

3.2.5.1 newline

newlines causes the current line of the file being written to, to go to the next line.

write(*newline*)

3.2.5.2 tab

tab causes a tab to be inserted at the current position of the file being written to.

write(*tab*)

3.2.6 read

read specifies the location to read text from a text file. The location grammar is defined in Section 3.2.1. Text that is read can be stored within an *identifier* with the becomes (Section 3.1.9.2) operator. Multiple column data must be stored within an Array.

read(*location*)

3.2.7 trim

trim statement defines how to handle whitespace surrounding a column. If trim is set to *true*, all leading and trailing whitespace is omitted. If trim is set to *false*, a column is left intact in its original form.

trim(*boolean*)

3.3 Statements

3.3.1 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.

if (*expression*) {

```
    $statement_1$;
    $statement_2$;
    .
    .
    .
    $statement_n$;
} else {
    $statement_1$;
    $statement_2$;
    .
    .
    .
    $statement_n$;
}
```

where n is the total number of *statements* and . represents the existence of multiple statements.

3.3.2 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.

```
while (expression) {
    $statement_1$;
    $statement_2$;
    .
    .
    .
    $statement_n$;
}
```

where n is the total number of *statements* and . represents the existence of multiple statements.

3.4 File Statements

File statements are evaluated using a . syntax. The . is punctuation used to separate the File *identifier* from either the *read*, *write*, *trim*, or *printF* functions.

Each function is performed on the text file represented by the File *identifier*. The statement syntax for each is:

identifier becomes *File_identifier*.read(*location*);

where *File_identifier* represents a filename and location and *identifier* represents a *String*, *int*, or array. If whitespace surrounds an integer in a column and *int* is the assignee, whitespace is implicitly omitted.

File_identifier.write(*expression*);
File_identifier.trim(*boolean*);
File_identifier.printF(*location*);

3.5 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.

3.5.1 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.

3.5.2 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) {
    statement₁;
    statement₂;
    .
    .
    .
```

    statement$_n$;
}

where n is the total number of *statements* and each *statement* is executed if var1 is greater than 100.

3.5.3 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 satisfied. An example logical expression is:

if(var1 == 50 or var2 > 100) {
    statement$_1$;
    statement$_2$;
    .
    .
    .
    statement$_n$;
}
where n is the total number of *statements* and each *statement* is executed if either var1 is equal to 50 or var2 is greater than 100.

4 Project Plan

4.1 Process

The development of the *Petros* language was conducted using research, testing, and concept refinement. The beginning stages of language development entailed deriving a language concept that was both practical and useful. Extensive research was done to understand the lexer, parser, and abstract syntax tree of a programming language. Language components were added and tested at every iteration. As *Petros* began to take shape, changes were made to the initial fundamental design until a final product was realized.

4.2 Programming Style

The *Petros* language was designed to mirror many aspects of the Java programming language. *Petros* was written using ANTLR in combination with Java. Therefore, Java conventions were followed during the writing of *Petros's* underlying system files. The various programming guidelines were used:

- Classes and methods are very structurally organized
- An extra space is inserted before an opening parenthesis and after a closing parenthesis for control structure statements

- No extra space is inserted before an opening parenthesis for method statements
- Local and Global variables are typically defined at the top of their code block
- Error states are checked and the program is exited if satisfied with an error message
- Methods are used were appropriate to facilitate organization and clarity of code
- Method and variables are mixed case and start with a lower-case letter
- Open braces are placed on the same line as the declaration
- Closing braces are placed on an individual line
- If/elseif/else and for loops do not include braces if they execute a single statement
- ANTLR token names consist of all capital letters in the lexer
- Non-terminal names in the parser are lowercase
- Indentation consists of 4 spaces

4.3 Software Development Environment

The *Petros* programming language was built using:

- Linux operating system
- ANTLR 2.7.7
- Vi text editor
- ssh

4.4 Project Timeline

1. June 3rd, Submitted Proposal
2. June 17th, Error free lexer and parser
3. June 18th, Submitted LRM
4. July 10th, First version of AST
5. July 15th, First version of tree walker
6. August 4th, Final AST and Java class Files
7. August 4-11, Testing and final report
8. August 11th, Final Report Submitted

5 Architectural Design

Petros is an interpreted programming language. The architectural diagram is illustrated in Figure 1. The Main class is responsible for instantiating the lexer, parser, and tree walker and passing the *Petros* program source file to the lexer. The lexer simply differentiates the various tokens, whitespace, and comments. These elements feed the parser which then checks the organization of tokens for syntactic errors. If a program is not syntactically correct, an error message is generated and the program terminates.

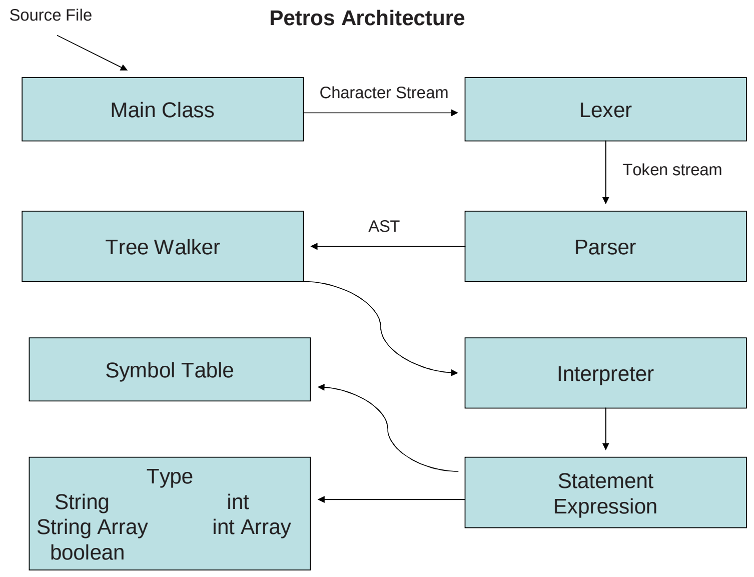

Fig. 1. Petros Architectural Design

Once a program is successfully parsed and an abstract syntax tree (AST) is generated by the parser, the static semantic analysis phase begins. At this point, the AST is walked as necessary by the tree walker from which various functions and classes are invoked to complete a task requested by the *Petros* program. The symbol table is responsible for storing the type of a particular data type as well its token for identification. Once an expression or statement has retrieved type and token information from the symbol table, it can retrieve values from one of the various type class children (i.e., int, String, File, etc). Statements represent a parent tree node and expressions a tree branch. Essentially, expressions are responsible for carrying out program functionality while statements direct traffic.

6 Test Plan and Example Programs

This section presents the tests used to prove the functionality of each language component. Tests aimed to prove that each component could be used in the intended fashion. Tests were broken into five different suites. Each suite is presented in the proceeding sections. Each suite proved *Petros's* functionality to be correct.

6.1 Text File Used For Test Suites

To facilitate the proof of language functionality, we define a text file that is intentionally arranged into six rows and six columns separated by three distinct delimiters (e.g., @ , .). The first three rows are string based data and the final three rows are integer based. The existence of both strings and integers provides an environment to test both *String* and *int* data types. Well-defined columns enable the proof of the language's fundamental principle, extraction of text from a file by location.

```
this is line 1, after comma 2. after period 3@ after at 4, after comma
     5. after period 6@ after at
this is line 2, after comma 2. after period 3@ after at 4, after comma
     5. after period 6@ after at
this is line 3, after comma 2. after period 3@ after at 4, after comma
     5. after period 6@ after at
1, 2. 3@ 4, 5. 6@
10, 20. 30@ 40, 50. 60@
100, 200. 300@ 400, 500. 600@
```

6.2 *Identifier* and *print* Test Suite

Identifiers and data types are the building blocks of the *Petros* programming language. This test case proves *String*, *StringArray*, *int*, and *intArray* ability to accurately initialize, print, and reference their content in all allowable ways. The program begins by initializing each variable and showing that arrays can be built in sequential order. After initialization, each variable's contents are printed to the screen. At this point, variables are assigned to each other and of particular interest is the ability to assign an entire array to another array. Also, an array of some arbitrary index is assigned to the same array of another index. Of most significance is the ability to nest array indices within other array's as is shown by $xx[y] = yy[yy[1]]$;. Finally, the initialization of an array that is not in sequential order produces an error as shown by $aa[3]$ in the final line of the program.

```
String a;
String b;
StringArray aa;
StringArray bb;

int x;
int y;
intArray xx;
intArray yy;

a = "this␣is␣string␣a";
b = "this␣is␣string␣b";
aa[0] = "this␣is␣a␣index␣0";
aa[1] = "this␣is␣a␣index␣1";
bb[0] = "this␣is␣b␣index␣0";
bb[1] = "this␣is␣b␣index␣1";

x = 100;
y = 200;
xx[0] = 110;
xx[1] = 120;
yy[0] = 500;
yy[1] = 0;

print("a␣=␣"^a^"␣␣and␣b␣=␣"^b);
print("aa[0]␣=␣"^aa[0]^"␣␣and␣aa[1]␣=␣"^aa[1]);
print("bb[0]␣=␣"^bb[0]^"␣␣and␣bb[1]␣=␣"^bb[1]);
print("x␣=␣"^x^"␣␣and␣y␣=␣"^y);

a = b;
```

```
aa = bb;
aa[0] = aa[1];

print("a is now = "^a);
print("aa[0] is now = "^aa[0]^"  and aa[1] is now = "^aa[1]);

x = y;
y = 0;
xx[y] = yy[yy[1]];

print("x is now = "^x^"  and y is now = "^y^"  and xx[y] = yy[yy[1]] is 
    now = "^xx[y]);

bb[2] = bb[yy[1]];
print("bb[2] = bb[yy[1]] is now = "^bb[2]);

aa[3] = "this should yield an error";
```

6.2.1 *Identifier* and *print* Test Suite Output

```
a = this is string a  and b = this is string b
aa[0] = this is a index 0  and aa[1] = this is a index 1
bb[0] = this is b index 0  and bb[1] = this is b index 1
x = 100  and y = 200
a is now = this is string b
aa[0] is now = this is b index 1  and aa[1] is now = this is b index 1
x is now = 200  and y is now = 0  and xx[y] = yy[yy[1]] is now = 500
bb[2] = bb[yy[1]] is now = this is b index 0
array index 3 is out of bounds. given current state of variable, '2' is maximum index
```

6.3 Precedence and Arithmetic, Equality, Relational, and Logical Operators Test Suite

In this test case we examine the correctness of the various *Petros* operators and precedence. Expressions are defined in such a way to show that different precedence would provide different results. Precedence is also forced in the expression containing parenthesis. Logical and relational operators are evaluated through a cascade of if/else statements. Evaluations are designed to yield both true and false expressions. Finally, the *equal to* operator is shown to evaluate a string's equality to another string.

```
int a;
int b;
intArray aa;
intArray bb;

a = 2;
b = 10;
aa[0] = 100;
aa[1] = 150;
bb[0] = 500;
bb[1] = 0;

aa[2] = bb[0] - a * b + aa[0];
```

```
if (b > a) {
    print("*should print* aa[2] should equal 580 and it = "^aa[2]);
} else {
    print("should NOT print");
}

aa[2] = bb[0] - a * (b + aa[1]);

if (b < a) {
    print("should NOT print");
} else {
    print("*should print* aa[2] should equal 180 and it = "^aa[2]);
}

aa[2] = aa[bb[1]] / b + a;

if (bb[1] <= aa[1] and b >= bb[1]) {
    print("*should print* aa[2] should equal 12 and it = "^aa[2]);
}

if (bb[1] <= 100 and 50 < 40 or 20 > 10) {
    print("should print");
}

if ("this string" == "this string") {
    print("should print");
}

if ("this string" == "that string") {
    print("should NOT print");
} else {
    print("should print");
}

if (bb[0] != bb[1]) {
    print("should print");
}

if (bb[1] > a and b > a) {
    print("should NOT print");
}

if (bb[1] > a or b < a or aa[1] < b) {
    print("should NOT print");
}

if (bb[1] < a or b < a or aa[1] < b) {
    print("should print");
}

while (bb[1] < 5) {
    print("bb[1] = "^bb[1]);
    bb[1] = bb[1] + 1;
}
```

6.3.1 Precedence and Arithmetic, Equality, Relational, and Logical Operators Test Suite Output

```
*should print* aa[2] should equal 580 and it = 580
*should print* aa[2] should equal 180 and it = 180
*should print* aa[2] should equal 12 and it = 12
should print
should print
should print
should print
should print
bb[1] = 0
bb[1] = 1
bb[1] = 2
bb[1] = 3
bb[1] = 4
```

6.4 *File*, *delimit*, and *printF* Test Suite

In the test suites that follow, we evaluate the functionality of manipulating a text file. In this section we examine the ability to define a file, set and unset delimiters, and print from various locations within the text file. Print statements are defined to test all possible location scenarios and ensure their correctness. For example, a single row, a range of rows, a range of rows ending at the end of file, a single row and single column, a single row and range of columns, a range of rows and a range of columns, and a range of rows to the end of file and a range of columns to the end of line. All evaluations proved correct.

```
File file;

file = "/home/joe/antlr/antlr-2.7.7/text.txt";

file.printF(3|4);
print("**************************************");
file.delimit(",");
file.printF(3:1|4:2);
print("**************************************");
file.delimit("");
file.printF(3|4);
print("**************************************");
file.delimit(",", "@", ".");
file.printF(3:3|5:~);
print("**************************************");
file.printF(1:1);
print("**************************************");
file.printF(3:2|~:~);
print("**************************************");
file.printF(2:2|2:5);
print("**************************************");
file.printF(6);
```

6.4.1 *File*, *delimit*, and *printF* Test Suite Output

```
this is line 3, after comma 2. after period 3@ after at 4, after comma 5. after period 6@ a
1, 2. 3@ 4, 5. 6@
**************************************
this is line 3, after comma 2. after period 3@ after at 4
1, 2. 3@ 4
**************************************
this is line 3, after comma 2. after period 3@ after at 4, after comma 5. after period 6@ a
1, 2. 3@ 4, 5. 6@
```

```
**************************************
 after period 3@ after at 4, after comma 5. after period 6@ after at
 3@ 4, 5. 6@
 30@ 40, 50. 60@
**************************************
this is line 1
**************************************
 after comma 2. after period 3@ after at 4, after comma 5. after period 6@ after at
 2. 3@ 4, 5. 6@
 20. 30@ 40, 50. 60@
 200. 300@ 400, 500. 600@
**************************************
 after comma 2. after period 3@ after at 4, after comma 5
**************************************
100, 200. 300@ 400, 500. 600@
```

6.5 *read*, *becomes*, and Control Statement Test Suite

In this section we prove the functionality of a program's ability to extract data from a file, store it, and use it to provide statistics or knowledge. The program reads data from a text file and initializes both *String* and *int* as well as arrays for both. This is done to ensure as many test scenarios as possible are accounted for. Integer values read from the text file are used for arithmetic operations and relational evaluations. Also, integer values read from the file are used to control the *while* statement in some instances. Both trim and contains statements are used in this test suite. The ability to omit leading and trailing whitespace is demonstrated as well as the ability to turn trim on and then off. Finally, the ability to test both *String* and *int* data types for the containing of values is shown for both.

```
File file;

String a;
StringArray aa;

int x;
int i;
intArray xx;
intArray yy;

file = "/home/joe/antlr/antlr-2.7.7/text.txt";
file.delimit(",", ".", "@");

a becomes file.read(1);

print("a␣=␣"^a);

aa becomes file.read(2:2|3:5);

x becomes file.read(4:4);

print("x␣=␣"^x);

i = 0;
```

```
while (i < x) {
    print("aa["^i^"]␣=␣"^aa[i]);
    i = i+1;
}

file.trim(true);
aa becomes file.read(2:2|3:5);

i = 0;
while (i < x) {
    print("aa["^i^"]␣=␣"^aa[i]);
    i = i+1;
}


file.trim(false);
aa becomes file.read(2:2|3:5);

i = 0;
while (i < x) {
    print("aa["^i^"]␣=␣"^aa[i]);
    i = i+1;
}

print("**********************************");

xx becomes file.read(5:3|~:~);
yy becomes file.read(4);

i = 0;
x = 8;
while (i < x) {
    print("xx["^i^"]␣=␣"^xx[i]);
    i = i+1;
}

i = 0;
x = 6;
while (i < x) {
    print("yy["^i^"]␣=␣"^yy[i]);
    i = i+1;
}

print("**********************************");

x = xx[1] + yy[5];

print("x␣=␣"^x);

if (aa[3] contains yy[4]) {
    print("*should␣print*␣because␣aa[3]␣=␣"^aa[3]^"␣and␣it␣contains␣yy
        [4]␣=␣"^yy[4]);
}

if (xx[4] contains yy[2]) {
    print("*should␣print*␣because␣xx[4]␣=␣"^xx[4]^"␣and␣it␣contains␣yy
        [2]␣=␣"^yy[2]);
}

if (aa[3] contains aa[2]) {
    print("should␣NOT␣print");
} else {
    print("should␣print");
}
```

6.5.1 *read*, *becomes*, and Control Statement Test Suite Output

```
a = this is line 1
x = 4
aa[0] =  after comma 2
aa[1] =  after period 3
aa[2] =  after at 4
aa[3] =  after comma 5
aa[0] = after comma 2
aa[1] = after period 3
aa[2] = after at 4
aa[3] = after comma 5
aa[0] =  after comma 2
aa[1] =  after period 3
aa[2] =  after at 4
aa[3] =  after comma 5
***********************************
xx[0] = 30
xx[1] = 40
xx[2] = 50
xx[3] = 60
xx[4] = 300
xx[5] = 400
xx[6] = 500
xx[7] = 600
yy[0] = 1
yy[1] = 2
yy[2] = 3
yy[3] = 4
yy[4] = 5
yy[5] = 6
***********************************
x = 46
*should print* because aa[3] =  after comma 5 and it contains yy[4] = 5
*should print* because xx[4] = 300 and it contains yy[2] = 3
should print
```

6.6 *write* Test Suite

The final test suite proves the functionality of the *write* statement. In this test both *String* and *int* data types are used as well as arrays to write to a file. This test case involves two separate files being written to simultaneously. After some time, both *File identifiers* are writing to the same file. Functionality is proven to be correct. Also, both *newline* and *tab* functions are shown to operate properly. Final output to both text files are shown below.

```
File file;
File file2;

```

```
String a;
StringArray aa;

int x;
intArray xx;

file = "/home/joe/antlr/antlr-2.7.7/write.txt";
file2 = "/home/joe/antlr/antlr-2.7.7/write2.txt";

a = "this is string a";
aa[0] = "this is aa index 0";
aa[1] = "this is aa index 1";

x = 100;
xx[0] = 0;
xx[1] = 1;

file.write(a^" and x = "^x);
file.write(newline);
file.write(aa[0]^" and xx[0] = "^xx[0]);
file.write(tab);
file.write("after tab aa[1] = "^aa[1]);

file2.write(aa[0]);
file2.write(newline);
file2.write(xx[0]^" and "^xx[1]);
file2.write(newline);

file = "/home/joe/antlr/antlr-2.7.7/write2.txt";
file.write("this should be at the end of write2.txt, however written by 
    ""file"" ");
file2.write(newline);
file2.write("written by file2");
file.write(newline);
file.write("written by file");
```

6.6.1 *write* Test Suite Output

6.6.1.1 write.txt

```
this is string a and x = 100
this is aa index 0 and xx[0] = 0     after tab aa[1] = this is aa index 1
```

6.6.1.2 write.txt

```
this is aa index 0
0 and 1
this should be at the end of write2.txt, however written by "file"
written by file2
written by file
```

7 Lessons Learned

Designing a programming language was a very rewarding and educational experience. I believe I now have a much deeper understanding of how a language is built and why various design choices are made. Before I began this project

I had a very difficult time understanding how ANTLR controlled the various components. I was very unsure of how ANTLR and Java interacted to build an AST and ultimately, walk it. This required many hours of research and testing.

At first I attempted to understand and build everything at once. This was an egregious mistake as I found myself very confused due to the complexity and therefore, making minimal progress. I realized I must make an adjustment and made the decision to get the most basic aspect of my language working, the print statement. From this point, I added one piece at a time until I achieved a language that was capable of accomplishing the fundamental principle of its existence, text file manipulation.

My advice to future teams is to start early and small. In other words, take baby steps until the lexer, parser, AST, and tree walker are fully understood. Only then can any real progress be made.

A Source Code

A.1 petros.g

```
class PetrosParser extends Parser;
options { buildAST = true; k = 2; }
tokens { DECLS; STMT; }

{
    int error = 0;

    public void reportError(String s) {
        super.reportError(s);
        error++;
    }

    public void reportError(RecognitionException e) {
        super.reportError(e);
        error++;
    }

}

decls : (decl | stmt) (decl | stmt)* { #decls = #([DECLS, "DECLS"], #
    decls); } ;

decl : ( "intArray" | "int" | "StringArray" | "File" | "String" ) ID
    SEMI! ;

array : ID LBRACK^ (NUM | ID | array) RBRACK! ;

stmt : factor ASSIGN^ bool SEMI!
     | "if"^ LPAREN! bool RPAREN! stmt (options {greedy=true;}: "else"!
         stmt)?
     | "while"^ LPAREN! bool RPAREN! stmt
     | ID PERIOD^ (delimit | read | printf | write | trim) SEMI!
     | "print"^ LPAREN! bool RPAREN! SEMI!
     | LBRACE! (stmt)* RBRACE! {#stmt = #([STMT, "STMT"], stmt); }
     | ID "becomes"^ stmt
     | SEMI
     ;

delimit : "delimit"^ LPAREN! delimArgs RPAREN! ;
delimArgs : STRING (COMMA^ STRING)* ;
read : "read"^ LOCATION ;
printf : "printF"^ LOCATION ;
write : "write"^ LPAREN! (join | "tab" | "newline") RPAREN! ;
trim : "trim"^ LPAREN! bool RPAREN! ;

bool     : equality (("and"^ | "or"^ | "contains"^) equality)* ;
equality : rel ((EQ^ | NE^) rel)* ;
rel      : expr ((LT^ | LE^ | GT^ | GE^) expr)* ;
expr     : term ((PLUS^ | MINUS^) term)* ;
term     : join ((MULT^ | DIV^) join)* ;
join     : factor (CONCAT^ factor)* ;
factor   : LPAREN! bool RPAREN! | NUM | STRING | ID | array | "true" | "
    false" ;



class PetrosLexer extends Lexer;
options { testLiterals = false; k = 2; charVocabulary = '\11'..'\177'; }

{
    int error = 0;
    public void reportError (String s) {
        super.reportError(s);
        error++;
    }

    public void reportError (RecognitionException e) {
        super.reportError(e);
        error++;
    }

}

WHITESPACE : (' ' | '\t' | '\f' | ( '\r' | '\n' ) { newline(); } )+ {
    $setType(Token.SKIP); } ;

NUM : ('0'..'9')+ ;

ASSIGN : '='     ;
EQ :     "=="    ;
NE :     "!="    ;
LT :     '<'     ;
LE :     "<="    ;
GT :     '>'     ;
GE :     ">="    ;
LBRACE : '{'     ;
RBRACE : '}'     ;
LBRACK : '['     ;
RBRACK : ']'     ;
SEMI :   ';'     ;
COLON :  ':'     ;
COMMA :  ','     ;
PERIOD : '.'     ;
SQUOTE : '\''    ;
DQUOTE : '"'     ;
LPAREN : '('     ;
RPAREN : ')'     ;
BAR :    '|'     ;
PLUS :   '+'     ;
MINUS :  '-'     ;
MULT :   '*'     ;
DIV :    '/'     ;
CONCAT : '^'     ;
END :    '~'     ;

STRING : DQUOTE! ( (~'"') | ( DQUOTE! DQUOTE ) )* DQUOTE! ;

LOCATION : LPAREN! NUM ( (COLON NUM (BAR (NUM | END) (COLON (NUM | END)
    )?)? ) | (BAR (NUM | END) (COLON (NUM | END) )?) )? RPAREN! ;

ID options {testLiterals=true;} : ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'
    ..'Z' | '0'..'9')* ;

COMMENT : "//" (~('\n' | '\r'))* { $setType(Token.SKIP); } ;

MULTICOMMENT : "/*" ( options { generateAmbigWarnings=false; }
                       : { LA(2)!='/' }? '*'
                       | '\r' '\n' { newline(); }
                       | '\r' { newline(); }
                       | '\n' {newline(); }
                       | ~('*' | '\n' | '\r')
                      )*
                      "*/" { $setType(Token.SKIP);}
                      ;

{
import java.io.File;
}
```

```
class PetrosWalker extends TreeParser;

{
    SymbolTable top = null;
    int scope = 0;
}

program returns [Stmt s]
{
 s = null; Type t = null; top = new SymbolTable(); }
  : #(DECLS (t=type ID { top.put(#ID.getText(), t, scope); } )*
s=stmts ) ;

type returns [Type t]
{ t = null; }
    : ( "bool" { t = Type.Bool; }
    | "intArray" { t = new IntA(); }
    | "int" { t = new Int(); }
    | "StringArray" { t = new StringA(); }
    | "File" { t = new FIle(); }
    | "String" { t = new STring(); }
    ) ;

stmts returns [Stmt s]
{ s = null; Stmt s1; }
    : (s=stmt)* ;

stmt returns [Stmt s]
{ Expr e1, e2, e = null; s = null; Stmt s1, s2; }

    : #(ASSIGN  e1=expr e2=expr
        { if (e1.getType().getName().equals("File") && e2.getType().
            getName().equals("String")) {
            File file = new File(e2.getType().toString());
            e2 = new Expr("File", new FIle(file));
          }

          if ((e1 != null && e2 != null)) { // && (e1 instanceof Id)) {

            if (e1.getType().getName().equals("intA")) {

            } else if (e1.getType().getName().equals("StringA")) {

            }

            if (!top.set(e1, e2))
                top.error("invalid types");
          } else
            top.error("assignment statement failed");

        }
    )

    | #("becomes" e1=expr s1=stmt
        { e2=null;

          if ((e1 != null && s1 != null) && (e1 instanceof Id)) {
            String v;

            if (e1.getType().getName().equals("int")) {
                v = (String) s1.vText.elementAt(0);
                v = v.trim();
                e2 = new Expr(v, new Int(Integer.parseInt(v)));
            } else if (e1.getType().getName().equals("String")) {
                v = (String) s1.vText.elementAt(0);
                e2 = new Expr(v, new STring(v));
            } else if (e1.getType().getName().equals("StringA") || e1.
                getType().getName().equals("intA")) {
            } else {
                System.out.println("cannot save as Type File when 
                    reading from a file --- must be int or String");
                System.exit(1);
            }

            if (e1.getType().getName().equals("StringA") || e1.getType()
                .getName().equals("intA"))
                top.set2(e1.getToken(), s1.vText);
            else
                if (!top.set(e1, e2))
                    top.error("invalid types at becomes");

        } else
                top.error("becomes statement failed");
    }
)

| #("while" whileexpr:. whilestmt:.
    { e1 = expr(#whileexpr);
      if (e1 != null) {
        while (e1.getToken().equals("true")) {
        s = stmt(#whilestmt);
        e1 = expr(#whileexpr);
        }
      } else {
        top.error("while expression failed");
      }
    }
)

| #("if" e1=expr thenp:. (elsep:.)?)
    { if ((e1 != null) && (thenp != null) && (e1.getToken().equals("
        true"))) {
        s = stmt(#thenp);
      } else if (null != elsep) {
        s=stmt(#elsep);
      }
    }

| #("print" e1=expr
    {
      if (e1 != null){
            if (e1.getType().getName().equals("int")) {
                System.out.println(e1.getType().toString());
            } else if (e1.getType().getName().equals("String")) {
                System.out.println(e1.getType().toString());
            } else if (e1.getType().getName().equals("StringA")) {
                StringA sA = (StringA) e1.getType();
                String ok = (String) sA.stringValues.elementAt(e1.
                    arrayIndex);
                System.out.println(ok);
            } else if (e1.getType().getName().equals("intA")) {
                IntA iA = (IntA) e1.getType();
                Object obj = iA.stringValues.elementAt(e1.arrayIndex
                    );
                String ok = obj.toString();
                System.out.println(ok);
            }
      } else {
        top.error("error with print statement");
      }
    }
)

| #(PERIOD e1=expr e2=expr
    { if (!e2.getType().getName().equals("String")) {
```

```
                System.out.println("invalid␣location␣specification");
                System.exit(1);
            }

            STring st = (STring) e2.getType();
            s = new Stmt();

            if (e1.getType().getName().equals("File") && st.getStatus().
                equals("delimit")) {
                FIle file = (FIle) e1.getType();
                file.setDelimit(e2.getType().toString());
            } else if (e1.getType().getName().equals("File") && st.
                getStatus().equals("read")) {
                FIle file = (FIle) e1.getType();
                s.vText = file.read(e2.getType().toString());
            } else if (e1.getType().getName().equals("File") && st.
                getStatus().equals("printF")) {
                FIle file = (FIle) e1.getType();
                file.print(e2.getType().toString());
            } else if (e1.getType().getName().equals("File") && st.
                getStatus().equals("write")) {
                FIle file = (FIle) e1.getType();
                STring sT = (STring) e2.getType();

                if (sT.newLine)
                    file.write("", true, false);
                else if (sT.tab)
                    file.write("", false, true);
                else
                    file.write(e2.getType().toString(), false, false);
            } else if (e1.getType().getName().equals("File") && st.
                getStatus().equals("trim")) {
                FIle file = (FIle) e1.getType();

                if (e2.getType().toString().equals("true")) {
                    file.trim = true;
                } else
                    file.trim = false;
            } else {
                System.out.println("invalid␣identifier␣−−␣must␣be␣type␣
                    File");
                System.exit(1);
            }

        }
    )

    | #(STMT stmt:.
        { s=stmts(#stmt); }
    ) ;

expr returns [Expr e]
    { Expr a, b; e = null; }

    : #(EQ          a=expr b=expr
        { if ((a.getType().getName().equals("String") || a.getType().
            getName().equals("StringA")) && (b.getType().getName().
            equals("String") ||
            b.getType().getName().equals("StringA"))) e = a.logic(7,b);
                else e = a.logic(1,b); } )
    | #(NE          a=expr b=expr { e = a.logic(2,b);      } )
    | #(LT          a=expr b=expr { e = a.logic(3,b);      } )
    | #(LE          a=expr b=expr { e = a.logic(4,b);      } )
    | #(GT          a=expr b=expr { e = a.logic(5,b);      } )
    | #(GE          a=expr b=expr { e = a.logic(6,b);      } )
    | #("contains"  a=expr b=expr { e = a.cont(b);         } )
    | #("and"       a=expr b=expr { e = a.boollogic(1,b); } )
    | #("or"        a=expr b=expr { e = a.boollogic(2,b); } )
| #(CONCAT      a=expr b=expr { e = a.concat(b);       } )
| #(PLUS        a=expr b=expr { e = a.arith(1,b);      } )
| #(MINUS       a=expr b=expr { e = a.arith(2,b);      } )
| #(MULT        a=expr b=expr { e = a.arith(3,b);      } )
| #(DIV         a=expr b=expr { e = a.arith(4,b);      } )
| #(COMMA       a=expr b=expr { e = a.delimit(b);      } )
| "true"                      { e = Constant.True;     }
| "false"                     { e = Constant.False;    }
| NUM           { String s=#NUM.getText(); e = new Expr(s, new Int(
    Integer.parseInt(s))); }
| STRING        { String s=#STRING.getText(); e = new Expr(s, new
    STring(s)); }
| LOCATION      { String s=#LOCATION.getText(); e = new Expr(s, new
    STring(s)); }
| #("delimit" e=expr
    { STring st = (STring) e.getType();
      st.setStatus("delimit");
    }
)
| #("read"  e=expr
    { STring st = (STring) e.getType();
      st.setStatus("read");
    }
)
| #("printF"  e=expr
    { STring st = (STring) e.getType();
      st.setStatus("printF");
    }
)
| #("write"  e=expr
    { if (e.getType().getName().equals("String")) {
          STring st = (STring) e.getType();
          st.setStatus("write");
      } else if (e.getType().getName().equals("StringA")) {
          StringA sA = (StringA) e.getType();
          String s = (String) sA.stringValues.elementAt(e.arrayIndex
              );

          STring st = new STring(s);
          st.setStatus("write");
          e = new Expr(s, st);
      } else if (e.getType().getName().equals("int")) {
          String s = e.getType().toString();
          STring st = new STring(s);
          st.setStatus("write");
          e = new Expr(s, st);
      } else if (e.getType().getName().equals("IntA")) {
          IntA iA = (IntA) e.getType();
          Object obj = iA.stringValues.elementAt(e.arrayIndex);
          String s = obj.toString();

          STring st = new STring(s);
          st.setStatus("write");
          e = new Expr(s, st);
      }
    }
)
| "tab"
    {
      String s = "TaBBB";
      STring st = new STring(s);
      st.setTab(true);
      e = new Expr(s, st);
    }
| "newline"
    {
      String s = "NeWlInEEE";
      STring st = new STring(s);
```

```
          st.setNewLine(true);
          e = new Expr(s, st);
        }
    | #("trim"   e=expr
        {
          String s = e.getToken();
          STring st = new STring(s);
          st.setStatus("trim");
          e = new Expr(s, st);
        }
    )
    | #(RBRACK  a=expr  b=expr    { e = a.array(b); } )
    | #(LBRACK  a=expr  b=expr
        { if (a != null && b != null) {
            if (a.getType().getName().equals("intA") || a.getType().
                getName().equals("StringA"))
                e = a.array(b);
            else {
                System.out.println("type␣mismatch␣−−␣identifier␣is␣not␣
                    an␣array␣type");
                System.exit(1);
            }
          }
        }
    )
    | #(ID
        { Id i = top.get(#ID.getText());

          if (i == null)
            System.out.print(#ID.getText()+"undeclared");

          e = i;
        }
    ) ;
```

A.2 Petros.java

```
/*
 * Main.java
 *
 * Created on July 12, 2008, 4:49 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

/**
 *
 * @author joe
 */

public class Petros {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        try {

            String file = args[0];
            if (!file.substring(file.length()-4, file.length()).equals("
                .pet")) {
                System.err.println("incorrect file type");
                System.exit(1);
            }
            FileInputStream filename = new FileInputStream(file);
            DataInputStream input = new DataInputStream(filename);
            PetrosLexer lexer = new PetrosLexer(input);
            PetrosParser parser = new PetrosParser(lexer);
//              parser.program();
            parser.decls();

            if (lexer.error > 0 || parser.error > 0) {
                System.err.println("could not parse program");
                System.exit(1);
            }

            CommonAST parseTree = (CommonAST) parser.getAST();
            PetrosWalker walker = new PetrosWalker();
            Stmt s = walker.program(parseTree);
        } catch (RecognitionException e) {
            System.err.println ("Recognition exception: "+e);
        } catch (TokenStreamException e) {
            System.err.println ("TokenStream exception: "+e);
        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("an error with DataInputStream");
            System.exit(1);
        }
    }

}
```

A.3 SymbolTable.java

```
import java.util.*;
import java.lang.Runtime.*;
import java.io.*;

public class SymbolTable {
    private Hashtable table;
    protected SymbolTable outer;

    public SymbolTable() {
        table = new Hashtable();
        outer = null;
    }

    public Id get(String token) {
        Id id = (Id) (table.get(token));
        if (id != null) {
            return id;
        }
        return null;
    }

    public void put(String token, Type t, int scope) {
        if (get(token) == null) {
            table.put(token, new Id(token, t, scope));
        } else {
            System.err.println(token+" is already defined");
            System.exit(1);
        }
    }

    public boolean checkType(Type t, Type e) {
        if (t.getName().equals("File") && e.getName().equals("String"))
            return true;
        else if (t.getName().equals("StringA") && e.getName().equals("
            String"))
            return true;
        else if (t.getName().equals("intA") && e.getName().equals("int")
            )
            return true;
        else
            return t.getName().equals(e.getName());
    }

    public void error(String s) {
        System.out.println(s);
        System.exit(0);
    }

    public String getType(Type t) {
        return t.getName();
    }

    public boolean set(Expr e1, Expr e) {
        Id id = (Id) get(e1.getToken());
        boolean same = checkType(id.getType(), e.getType());

        if (same) {
            String value = getType(id.getType());

            if (value.equals("int")) {
                Int i = (Int) id.getType();
                Int m = (Int) e.getType();
                i.setValue(m.getValue());
                return true;
            } else if (value.equals("String")) {
                STring x = (STring) id.getType();
                STring y = (STring) e.getType();
                x.setValue(y.getValue());
                return true;
            } else if (value.equals("File")) {
                FIle a = (FIle) id.getType();
                FIle b = (FIle) e.getType();

                try {
                    a.setFile(b.file);
                    a.setName(b.name);
                } catch (NullPointerException npe) {
                    System.out.println("null pointer exception at set(-
                        File-) symbol table");
                }
                return true;
            } else if (value.equals("StringA")) {
                if (e.getType().getName().equals("String")) {
                    StringA x = (StringA) id.getType();
                    STring y = (STring) e.getType();

                    x.setValue(y.getValue(), e1.arrayIndex);
                    return true;
                } else if (e.getType().getName().equals("StringA")) {
                    StringA x = (StringA) id.getType();
                    StringA y = (StringA) e.getType();

                    if (e.arrayIndex == -1)
                        x.setVector(y.stringValues);
                    else {
                        String temp = (String) y.stringValues.elementAt(
                            e.arrayIndex);
                        x.setValue(temp, e1.arrayIndex);
                    }

                    return true;
                } else {
                    System.out.println("type␣mismatch␣for␣String␣array")
                        ;
                    System.exit(1);
                }
            } else if (value.equals("intA")) {
                if (e.getType().getName().equals("int")) {
                    IntA x = (IntA) id.getType();
                    Int y = (Int) e.getType();

                    x.setValue(y.getValue(), e1.arrayIndex);
                    return true;
                } else if (e.getType().getName().equals("intA")) {
                    IntA x = (IntA) id.getType();
                    IntA y = (IntA) e.getType();

                    if (e.arrayIndex == -1)
                        x.setAlreadyIntVector(y.stringValues);
                    else {
                        Object obj = y.stringValues.elementAt(e.
                            arrayIndex);
                        x.setValue(Integer.parseInt(obj.toString()), e1.
                            arrayIndex);
                    }
                    return true;
                } else {
                    System.out.println("type␣mismatch␣for␣int␣array");
                    System.exit(1);
                }
            } else {
                System.out.println("types␣couldnt␣be␣set");
                System.exit(1);
            }
        }
        return false;
    }

    public void set2(String token, Vector vText) {
        Id id = (Id) get(token);

        if (id.getType().getName().equals("StringA")) {
            StringA x = (StringA) id.getType();
            x.setVector(vText);
        } else if (id.getType().getName().equals("intA")) {
            IntA x = (IntA) id.getType();
            x.setVector(vText);
        } else {
            System.out.println("i␣think␣this␣state␣is␣impossible,␣but␣
                neither␣StringA␣or␣IntA␣was␣satisfied␣in␣symbol␣table␣
                set2()");
            System.exit(1);
        }
    }

    public void setIntVector(String token, Expr e) {

    }

    public boolean hasEls() {
        return table.isEmpty();
    }

    public String toString() {
        String s = ""; String key = "";
        boolean debug = true;

        if (debug) {
            Enumeration keys = table.keys();

            while (keys.hasMoreElements()) {
                key = (String) keys.nextElement();
                Id id = (Id) get(key);
                System.out.println(key+"␣"+id.getType().toString());
            }
        } else {
            s = table.toString();
        }
        return s;
    }
}
```

A.4 Stmt.java

```
import java.util.*;

public class Stmt {
//    public String text;
    public Vector vText;

    public Stmt() {

    }
}
```

A.5 Type.java

```
public class Type {
    public String name = "␣";

    public static final Type Bool = new Type("bool");

    public Type(String s) {
        name = s;
    }

    public String getName() {
        return this.name;
    }
}
```

A.6 Id.java

```
public class Id extends Expr {
    public String s;
    public Type type;
    public int scope;

    public Id(String id, Type p, int scope) {
        super(id, p);
        this.scope = scope;
    }

    public int getScope() {
        return this.scope;
    }

}
```

A.7 Expr.java

```
public class Expr {
    public String s;
    public Type type;
    private Int i, j;
    private STring x, y;
    public int arrayIndex = −1;

    Expr(String S, Type p) {
        s = S;
        type = p;
    }

    public String getToken() {
        return this.s;
    }

    public Type getType() {
        return this.type;
    }

    public Expr arith(int op, Expr e) {
        IntA x, y;
        int a = 0, b = 0;
        Object obj;

        if (this.type.getName().equals("int")) {
            i = (Int) this.type;
            a = i.getValue();
        } else if (this.type.getName().equals("intA")) {
            x = (IntA) this.type;
            obj = x.stringValues.elementAt(this.arrayIndex);
            a = Integer.parseInt(obj.toString());
        } else {
            System.out.println("type␣mismatch␣at␣Expr␣arith␣−−␣cannot␣
                equal:␣"+this.type.getName());
            System.exit(1);
        }

        if (e.getType().getName().equals("int")) {
            j = (Int) e.getType();
            b = j.getValue();
        } else if (e.getType().getName().equals("intA")) {
            y = (IntA) e.getType();
            obj = y.stringValues.elementAt(e.arrayIndex);
            b = Integer.parseInt(obj.toString());
        } else {
            System.out.println("type␣mismatch␣at␣Expr␣arith␣−−␣2nd␣term"
                );
            System.exit(1);
        }

        int c = 0;
        switch(op) {
            case 1:
                c = a + b;
                return new Expr(c+"", new Int(c));
            case 2:
                c = a - b;
                return new Expr(c+"", new Int(c));
            case 3:
                c = a * b;
                return new Expr(c+"", new Int(c));
            case 4:
                c = a / b;
                return new Expr(c+"", new Int(c));
        }
        return null;
    }

    public Expr concat(Expr e) {
        StringA sA;
        String temp;
        IntA iA;
        String a = "", b = "";
        Object obj;

        if (this.type.getName().equals("int"))
            a = this.getType().toString();
        else if (this.type.getName().equals("intA")) {
            iA = (IntA) this.type;
            obj = iA.stringValues.elementAt(this.arrayIndex);
            a = obj.toString();
        } else if (this.type.getName().equals("String")) {
            a = this.getType().toString();
        } else if (this.type.getName().equals("StringA")) {
            sA = (StringA) this.type;
            a = (String) sA.stringValues.elementAt(this.arrayIndex);
        }

        if (e.getType().getName().equals("int"))
            b = e.getType().toString();
        else if (e.getType().getName().equals("intA")) {
            iA = (IntA) e.getType();
            obj = iA.stringValues.elementAt(e.arrayIndex);
            b = obj.toString();
        } else if (e.getType().getName().equals("String")) {
            b = e.getType().toString();
        } else if (e.getType().getName().equals("StringA")) {
            sA = (StringA) e.getType();
            b = (String) sA.stringValues.elementAt(e.arrayIndex);
        }

        String c;
        c = a.concat(b);
        return new Expr(c+"", new STring(c));
    }

    public Expr cont(Expr e) {
        StringA sA;
        String temp;
        IntA iA;
        String a = "", b = "";
        Object obj;

        if (this.type.getName().equals("int"))
            a = this.getType().toString();
        else if (this.type.getName().equals("intA")) {
            iA = (IntA) this.type;
            obj = iA.stringValues.elementAt(this.arrayIndex);
            a = obj.toString();
    } else if (this.type.getName().equals("String")) {
        a = this.getType().toString();
    } else if (this.type.getName().equals("StringA")) {
        sA = (StringA) this.type;
        a = (String) sA.stringValues.elementAt(this.arrayIndex);
    }

    if (e.getType().getName().equals("int"))
        b = e.getType().toString();
    else if (e.getType().getName().equals("intA")) {
        iA = (IntA) e.getType();
        obj = iA.stringValues.elementAt(e.arrayIndex);
        b = obj.toString();
    } else if (e.getType().getName().equals("String")) {
        b = e.getType().toString();
    } else if (e.getType().getName().equals("StringA")) {
        sA = (StringA) e.getType();
        b = (String) sA.stringValues.elementAt(e.arrayIndex);
    }

    if (a.indexOf(b) != -1)
        return Constant.True;
    else
        return Constant.False;
}

public Expr array(Expr e) {
    Expr e2 = null;
    Int i;
    IntA iA, iA2;
    StringA sA;
    Object obj;

    if (e.getType().getName().equals("int")) {

        if (this.getType().getName().equals("intA")) {
            iA = (IntA) this.type;
            i = (Int) e.getType();

            e2 = new Expr(this.getToken(), iA);
            e2.arrayIndex = i.getValue();
            iA.arrayIndex = e2.arrayIndex;
            return e2;
        } else {
            sA = (StringA) this.type;
            i = (Int) e.getType();

            e2 = new Expr(this.getToken(), sA);
            e2.arrayIndex = i.getValue();
            sA.arrayIndex = e2.arrayIndex;
            return e2;
        }

    } else if (e.getType().getName().equals("intA")) {
        if (this.getType().getName().equals("intA")) {
            iA = (IntA) this.type;
            iA2 = (IntA) e.getType();

            obj = iA2.stringValues.elementAt(iA2.arrayIndex);
            e2 = new Expr(this.getToken(), iA);
            e2.arrayIndex = Integer.parseInt(obj.toString());
            iA.arrayIndex = e2.arrayIndex;
            return e2;
        } else {
            sA = (StringA) this.type;
            iA2 = (IntA) e.getType();

            obj = iA2.stringValues.elementAt(iA2.arrayIndex);
```

```
                e2 = new Expr(this.getToken(), sA);
                e2.arrayIndex = Integer.parseInt(obj.toString());
                sA.arrayIndex = e2.arrayIndex;
                return e2;
            }

        } else {
            System.out.println("invalid type for index -- must be of 
                type int or intArray");
            System.exit(1);
        }
        return e2;
    }

    public Expr delimit(Expr e) {
        String c;
        String temp = "";

        boolean end = this.getType().toString().endsWith("]");

        if (end) {
            int x = this.getType().toString().length();

            for (int i = 0; i < x; i++) {
                if ((i != 0) && (i != (x-1)))
                    temp = temp+this.getType().toString().charAt(i);
            }
        } else
            temp = this.getType().toString();

        c = "[".concat(temp).concat(e.getType().toString()).concat("]");

        STring st = new STring(c);
        return new Expr(c+"", st);
    }

    public Expr boollogic(int op, Expr e) {

        if (this.type.getName().equals("bool") && e.getType().getName().
            equals("bool")) {
            String s1 = this.s;
            String s2 = e.getToken();

            int c = 0;
            switch (op) {
                case 1:
                    if (s1.equals("true") && s2.equals("true"))
                        return Constant.True;
                    else
                        return Constant.False;
                case 2:
                    if (s1.equals("true") || s2.equals("true"))
                        return Constant.True;
                    else
                        return Constant.False;
            }
        }
        return Constant.False;
    }

    public Expr logic(int op, Expr e) {
        IntA x, y;
        int a = -1, b = -2;
        StringA sA;
        String m = "", n = "";
        Object obj;

        if (this.type.getName().equals("int")) {
            i = (Int) this.type;
            a = i.getValue();
        } else if (this.type.getName().equals("intA")) {
            x = (IntA) this.type;
            obj = x.stringValues.elementAt(this.arrayIndex);
            a = Integer.parseInt(obj.toString());
        } else if (this.type.getName().equals("String")) {
            m = this.getType().toString();
        } else if (this.type.getName().equals("StringA")) {
            sA = (StringA) this.type;
            m = (String) sA.stringValues.elementAt(this.arrayIndex);
        }

        if (e.getType().getName().equals("int")) {
            j = (Int) e.getType();
            b = j.getValue();
        } else if (e.getType().getName().equals("intA")) {
            y = (IntA) e.getType();
            obj = y.stringValues.elementAt(e.arrayIndex);
            b = Integer.parseInt(obj.toString());
        } else if (e.getType().getName().equals("String")) {
            n = e.getType().toString();
        } else if (e.getType().getName().equals("StringA")) {
            sA = (StringA) e.getType();
            n = (String) sA.stringValues.elementAt(e.arrayIndex);
        }

            int c = 0;
            switch (op) {
                case 1:
                    if (a == b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 2:
                    if (a != b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 3:
                    if (a < b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 4:
                    if (a <= b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 5:
                    if (a > b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 6:
                    if (a >= b)
                        return Constant.True;
                    else
                        return Constant.False;
                case 7:
                    if (m.equals(n))
                        return Constant.True;
                    else
                        return Constant.False;
            }
        return null;
    }
}
```

A.8 FIle.java

```
import java.io.*;
import java.lang.*;
import java.util.*;

public class FIle extends Type {
    public File file;
    public String name, delimit;
    int[] row, column;
    boolean trim = false;

    public FIle() {
        super("File");
        this.name = "";
        this.file = null;
        this.delimit = null;
    }

    public FIle(File FILE) {
        super("File");
        this.file = FILE;
        this.name = FILE.getName();
        this.delimit = null;
    }

    public Vector read(String s) {
        Vector vText = new Vector(1000);
        String lineTemp = "";
        String[] tokens;

        parseLocation(s);

        try {
            DataInputStream in = new DataInputStream(new FileInputStream
                (this.file));

            for (int i = 0; i < row[0]; i++)
                lineTemp = in.readLine();

            if (column == null) {
                if (this.delimit == null)
                    if (trim == false)
                        vText.addElement(lineTemp);
                    else
                        vText.addElement(lineTemp.trim());
                else {
                    tokens = lineTemp.split(this.delimit);
                    for (int i = 0; i < tokens.length; i++)
                        if (trim == false)
                            vText.addElement(tokens[i]);
                        else
                            vText.addElement(tokens[i].trim());
                }

                if (row.length == 1)
                    return vText;
                else if (row[0] == row[1])
                    return vText;
                else {

                    for (int i = row[0]; i < row[1]; i++) {
                        lineTemp = in.readLine();

                        if (lineTemp == null)
                            return vText;
                    if (this.delimit == null)
                        if (trim == false)
                            vText.addElement(lineTemp);
                        else
                            vText.addElement(lineTemp.trim());
                    else {
                        tokens = lineTemp.split(this.delimit);
                        for (int j = 0; j < tokens.length; j++)
                            if (trim == false)
                                vText.addElement(lineTemp);
                            else
                                vText.addElement(lineTemp.trim());
                    }
                }
                return vText;
            }
        } else {    /*** column exists ***/
            tokens = lineTemp.split(this.delimit);

            if (row.length == 1 && column.length == 1) {
                if (trim == false)
                    vText.addElement(tokens[column[0]-1]);
                else
                    vText.addElement(tokens[column[0]-1].trim());
                return vText;
            } else if (row.length == 2 && column.length == 2) {

                if (row[0] == row[1] && column[0] == column[1]) {
                    if (trim == false)
                        vText.addElement(tokens[column[0]-1]);
                    else
                        vText.addElement(tokens[column[0]-1].trim())
                            ;
                return vText;
                } else if (row[0] == row[1]) {

                    if (column[1] == Integer.MAX_VALUE) {
                        for (int i = (column[0] - 1); i < (tokens.
                            length); i++)
                            if (trim == false)
                                vText.addElement(tokens[i]);
                            else
                                vText.addElement(tokens[i].trim());
                        return vText;
                    } else {
                        for (int i = (column[0] - 1); i < (column
                            [1]); i++)
                            if (trim == false)
                                vText.addElement(tokens[i]);
                            else
                                vText.addElement(tokens[i].trim());
                        return vText;
                    }
                } else {
                    if (column[0] == column[1])
                        if (trim == false)
                            vText.addElement(tokens[column[0]-1]);
                        else
                            vText.addElement(tokens[column[0]-1].
                                trim());
                    else
                        if (column[1] == Integer.MAX_VALUE)
                            for (int i = (column[0] - 1); i < (
                                tokens.length); i++)
                                if (trim == false)
                                    vText.addElement(tokens[i]);
                                else
```

```
                                    vText.addElement(tokens[i].trim
                                        ());
                        else
                            for (int i = (column[0]-1); i < (column
                                [1]); i++)
                                if (trim == false)
                                    vText.addElement(tokens[i]);
                                else
                                    vText.addElement(tokens[i].trim
                                        ());

                    for (int i = row[0]; i < row[1]; i++) {
                        lineTemp = in.readLine();

                        if (lineTemp == null)
                            return vText;

                        tokens = lineTemp.split(this.delimit);

                        if (column[1] == Integer.MAX_VALUE)
                            for (int j = (column[0] - 1); j < (
                                tokens.length); j++)
                                if (trim == false)
                                    vText.addElement(tokens[j]);
                                else
                                    vText.addElement(tokens[j].trim
                                        ());
                        else
                            for (int j = (column[0] - 1); j < (
                                column[1]); j++)
                                if (trim == false)
                                    vText.addElement(tokens[j]);
                                else
                                    vText.addElement(tokens[j].trim
                                        ());
                    }
                    return vText;
                }

            } else {
                System.out.println("invalid location specification 
                    -- if 1 row then 1 column and if 2 rows then 2 
                    columns");
                System.exit(1);
            }

        }
        in.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        System.out.println("DataIOTest: "+e);
        System.exit(1);
    } catch (IOException e) {
        e.printStackTrace();
        System.err.println("an error with DataInputStream");
        System.exit(1);
    }
    return vText;
}

public void write(String text, boolean newLine, boolean tab) {

    try {
        RandomAccessFile raf = new RandomAccessFile(this.file, "rw")
            ;

        raf.seek(this.file.length());
```

```
            if (newLine)
                raf.writeChar('\n');
            else if (tab)
                raf.writeBytes("␣␣␣␣␣");
            else
                raf.writeBytes(text);

            raf.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.err.println("\nFILE_NOT_FOUND\n");
            System.exit(1);
        }  catch (IOException e) {
            e.printStackTrace();
            System.err.println("\nIOException\n");
            System.exit(1);
        }

    }

    public void print(String s) {
        StringTokenizer st;
        Vector vText = new Vector(1000);
        String lineTemp = "", desiredText = "";
        String[] tokens;

        parseLocation(s);

        try {
            DataInputStream in = new DataInputStream(new FileInputStream
                (this.file));

            for (int i = 0; i < row[0]; i++)
                lineTemp = in.readLine();

            if (column == null) {
                if (row.length == 1) {
                    System.out.println(lineTemp);
                    return;
                } else if (row[0] == row[1]) {
                    System.out.println(lineTemp);
                    return;
                } else
                    System.out.println(lineTemp);

                for (int i = row[0]; i < row[1]; i++) {
                    lineTemp = in.readLine();

                    if (lineTemp == null)
                        return;

                    System.out.println(lineTemp);
                }

            } else {
                st = new StringTokenizer(lineTemp, this.delimit, true);
                int numberOfTokens;

                if (column.length == 1) {
                    if (column[0] == Integer.MAX_VALUE)
                        numberOfTokens = st.countTokens();
                    else
                        numberOfTokens = 2 * column[0] - 2;

                    for (int i = 0; i < numberOfTokens; i++)
                        st.nextToken();

                    desiredText = st.nextToken();
                    System.out.println(desiredText);
                    desiredText = "";

                    if (row.length == 1)
                        return;
/*
                    for (int i = row[0]; i < row[1]; i++) {
                        lineTemp = in.readLine();

                        if (lineTemp == null)
                            return;

                        st = new StringTokenizer(lineTemp, this.delimit,
                              true);

                        for (int j = 0; j < (numberOfTokens - 1); j++)
                            st.nextToken();

                        desiredText = st.nextToken();
                        System.out.println(desiredText);
                    }
*/
                } else {
                    if (column[1] == Integer.MAX_VALUE)
                        numberOfTokens = st.countTokens();
                    else
                        numberOfTokens = 2 * column[1] - 1;

                    for (int i = 0; i < (2 * column[0] - 2); i++)
                        st.nextToken();

                    for (int i = (2 * column[0] - 2); i < numberOfTokens
                        ; i++)
                        desiredText = desiredText+st.nextToken();

                    System.out.println(desiredText);
                    desiredText = "";

                    if (row[0] == row[1])
                        return;
                    else {

                        for (int i = row[0]; i < row[1]; i++) {
                            lineTemp = in.readLine();

                            if (lineTemp == null)
                                return;

                            st = new StringTokenizer(lineTemp, this.
                                delimit, true);

                            if (column[1] == Integer.MAX_VALUE)
                                numberOfTokens = st.countTokens();
                            else
                                numberOfTokens = 2 * column[1] - 1;

                            for (int j = 0; j < (2 * column[0] - 2); j
                                ++)
                                st.nextToken();

                            for (int j = (2 * column[0] - 2); j <
                                numberOfTokens; j++)
                                desiredText = desiredText+st.nextToken()
                                    ;

                            System.out.println(desiredText);
                            desiredText = "";
                        }

                    }

                }
            }
            in.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.out.println("DataIOTest: "+e);
            System.exit(1);
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("an error with DataInputStream");
            System.exit(1);
        }

    }

    public void parseLocation(String LOCATION) {

        String[] middle = LOCATION.split("\\|");
        String[] left, right;

        left = middle[0].split(":");

        if (middle.length == 2) {
            row = new int[2];
            right = middle[1].split(":");

            for (int i = 0; i < right.length; i++)
                if (right[i].equals("~"))
                    right[i] = Integer.toString(Integer.MAX_VALUE);

            if (right.length == 2) {
                column = new int[2];
                row[0] = Integer.parseInt(left[0]);
                row[1] = Integer.parseInt(right[0]);

                if (left.length == 1) {
                    column[0] = 1;
                    column[1] = Integer.parseInt(right[1]);
                } else {
                    column[0] = Integer.parseInt(left[1]);
                    column[1] = Integer.parseInt(right[1]);
                }
            } else {
                if (left.length == 2) {
                    System.out.println("invalid location --- beginning 
                        column is specified with no end column");
                    System.exit(1);
                } else {
                    column = null;
                    row[0] = Integer.parseInt(left[0]);
                    row[1] = Integer.parseInt(right[0]);
                }
            }
        } else {
            if (left.length == 2) {
                row = new int[1];
                column = new int[1];
                row[0] = Integer.parseInt(left[0]);
                column[0] = Integer.parseInt(left[1]);
            } else {
                row = new int[1];
                column = null;
                row[0] = Integer.parseInt(left[0]);
            }
        }

        if (column != null)
            if (row.length == 2 && column.length == 2)
                if (row[0] == 0 || row[1] == 0 || column[0] == 0 ||
                    column[1] == 0) {
                    System.out.println("invalid␣location␣--␣row␣or␣
                        column␣cannot␣=␣0");
                    System.exit(1);
                } else if (row[1] < row[0] || column[1] < column[0]) {
                    System.out.println("invalid␣location␣--␣ending␣
                        column␣and␣row␣bounds␣must␣be␣greater␣than␣begin
                        ␣bounds");
                    System.exit(1);
                }
             else if (row.length == 1 && column.length == 1)
                if (row[0] == 0 || column[0] == 0) {
                    System.out.println("invalid␣location␣--␣row␣or␣
                        column␣cannot␣=␣0");
                    System.exit(1);
                }
        else if (row.length == 2 && column == null)
            if (row[0] == 0 || row[1] == 0) {
                System.out.println("invalid␣location␣--␣row␣or␣column␣
                    cannot␣=␣0");
                System.exit(1);
            } else if (row[1] < row[0]) {
                System.out.println("invalid␣location␣--␣ending␣column␣
                    and␣row␣bounds␣must␣be␣greater␣than␣begin␣bounds");
                System.exit(1);
            }
    }

    public String toString() {
        return this.name;
    }

    public void setFile(File FILE) {
        this.file = FILE;
    }

    public void setName(String s) {
        this.name = s;
    }

    public void setDelimit(String s) {
        if (s.equals(""))
            this.delimit = null;
        else
            this.delimit = s;
    }
}
```

A.9 STring.java

```
public class STring extends Type {
    public String value;
    public String status;
    public boolean newLine = false, tab = false;
    public int arrayIndex;

    public STring() {
        super("String");
        this.value = "";
        this.status = "";
    }

    public STring(String value) {
        super("String");
        this.value = value;
        this.status = "";
    }

    public String toString() {
        return this.value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public String getValue() {
        return this.value;
    }

    public void setStatus(String s) {
        this.status = s;
    }

    public String getStatus() {
        return this.status;
    }

    public void setIndex(int index) {
        this.arrayIndex = index;
    }

    public void setNewLine(boolean nLine) {
        this.newLine = nLine;
    }

    public void setTab(boolean t) {
        this.tab = t;
    }

}
```

A.10 StringA.java

```
import java.util.*;

public class StringA extends Type {
    public String value;
    public String status;
    Vector stringValues;
    public int arrayIndex = -1;

    public StringA() {
        super("StringA");
        this.stringValues = new Vector();
    }

    public void setVector(Vector vText) {
        Vector v = (Vector) vText.clone();

        this.stringValues = v;
    }

    public String toString() {
        String a = "";
        return a;
    }

    public void setValue(String value, int index) {
        Vector vector2;

        if (index == this.stringValues.size()) {
            if (index == this.stringValues.size()) {
                this.stringValues.addElement(value);
                vector2 = (Vector) this.stringValues.clone();
                this.stringValues = vector2;
            }
        } else if (index > this.stringValues.size()) {
            System.out.println("array␣index␣"+index+"␣is␣out␣of␣bounds.␣
                given␣current␣state␣of␣variable,␣'"+this.stringValues.
                size()+"'␣is␣maximum␣index");
            System.exit(1);
        } else {
            this.stringValues.setElementAt(value, index);
            vector2 = (Vector) this.stringValues.clone();
            this.stringValues = vector2;
        }
    }

    public void setStatus(String s) {
        this.status = s;
    }

    public void setIndex(int index) {
        this.arrayIndex = index;
   }

}
```

A.11 Int.java

```
public class Int extends Type {
    public int value;
    public String status;
    public int arrayIndex;

    public Int() {
        super("int");
        this.value = 0;
    }

    public Int(int value) {
        super("int");
        this.value = value;
    }

    public String toString() {
        return ""+this.value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }

    public void setStatus(String s) {
        this.status = s;
    }

    public void setIndex(int index) {
        this.arrayIndex = index;
    }

}
```

A.12 IntA.java

```
import java.util.*;

public class IntA extends Type {
    public String value;
    public String status;
    Vector stringValues;
    public int arrayIndex = -1;

    public IntA() {
        super("intA");
        this.stringValues = new Vector();
    }

    public void setVector(Vector vText) {
        Vector v = (Vector) vText.clone();

        this.stringValues = v;

        for (int i = 0; i < this.stringValues.size(); i++) {
            String s = (String) this.stringValues.elementAt(i);
            s = s.trim();
            this.stringValues.setElementAt(Integer.parseInt(s), i);
        }
    }

    public void setAlreadyIntVector(Vector vText) {
        Vector v = vText;

        this.stringValues = v;
    }

    public String toString() {
        String a = "";
        return a;
    }

    public void setValue(int value, int index) {
        Vector vector2;

        if (index == this.stringValues.size()) {
            if (index == this.stringValues.size()) {
                this.stringValues.addElement(value);
                vector2 = (Vector) this.stringValues.clone();
                this.stringValues = vector2;
            }
        } else if (index > this.stringValues.size()) {
            System.out.println("array␣index␣"+index+"␣is␣out␣of␣bounds.␣
                given␣current␣state␣of␣variable,␣'"+this.stringValues.
                size()+"'␣is␣maximum␣index");
            System.exit(1);
        } else {
            this.stringValues.setElementAt(value, index);
            vector2 = (Vector) this.stringValues.clone();
            this.stringValues = vector2;
        }
    }

    public void setStatus(String s) {
        this.status = s;
    }

    public void setIndex(int index) {
        this.arrayIndex = index;
    }

}
```

A.13 Constant.java

```
public class Constant extends Expr {

    public Constant(String S, Type p) {
        super(S, p);
    }

    public static final Constant
        True = new Constant("true", Type.Bool), False = new Constant("
            false", Type.Bool);
}
```