Prof. Stephen Edwards
Programming Languages, and Translators
W4115
Fall 2005

DFile Language

By

Amrita Rajagopal
Anton Ushakov
Erwin Polio
Howie Vegter

Table of Contents

Chapter – 1 Introduction........6
- 1.1 Definition........6
  - 1.2 Background........6
  - 1.3 Goal........6
  - 1.4 Ease of Use........6
  - 1.5 Portability........7

Chapter 2 - Language Tutorial........8
- 2.1 Getting Started........8
- 2.2 Compiling a program.........8
- 2.3 Writing a Program........8

Chapter 3 - Language Reference Manual........11
- 3.1 Introduction........11
- 3.2. Syntax notation........11
- 3.3. Lexical conventions........11
  - 3.3.1 Comments........11
  - 3.3.2 Identifiers (Names)........11
  - 3.3.3 Keywords........12
  - 3.3.4 Constants........12
  - 3.3.4.1 Number constants........12
  - 3.3.4.1.1 Decimal constant........12
  - 3.3.4.2 String constants........13
- 3.4 Statements........13
  - 3.4.1 Expressions........13
  - 3.4.2 Identifier........13
  - 3.4.3 Constant........13
  - 3.4.4 String........13
- 3.5. Keywords........14
  - 3.5.1 If…Else........14
  - 3.5.1.1 Heading........14
  - 3.5.1.2 Footer........14
  - 3.5.1.3 ColNames........14
  - 3.5.1.2 For........15
  - 3.5.1.3 ForEach........15
  - 3.5.1.4 Delimiter........15
  - 3.5.1.5 Include........15
  - 3.5.1.6 Print........16
  - 3.5.1.7 Function........16
  - 3.5.1.8 Return........16
  - 3.5.1.9 isNum........16
  - 3.5.1.10 isDefined........17
- 3.6. Operators........17
  - 3.6.1. Logical Operators........17

3.6.1.1 expression || expression 17
3.6.1.2 expression && expression 17
3.6.2 Relational operators 18
3.6.2.1 expression < expression, expression <= expression 18
3.6.2.2 expression > expression, expression >= expression 18
3.6.3 Equality operators 18
3.6.3.1 expression == expression 18
3.6.3.2 expression != expression 18
3.6.4 Multiplicative 18
3.6.4.1 expression * expression 18
3.6.4.2 expression / expression 18
3.6.5 Additive operators 18
3.6.5.1 expression + expression 19
3.6.5.2 expression – expression 19
3.6.6. Assignment 19
3.6.6.1. lvalue = expression 19
3.6.6.2 = Operator 19
3.6.6.3 expression += expression 19
3.6.6.4 expression -= expression 19
3.6.6.5 expression *= expression 19
3.6.6.6 expression /= expression 19
3.6.6.7 expression &= expression 19
3.6.7 Concatenation 20
3.6.8 # (Pound) 20
3.6.9 ~= (Tilda equal) 20
3.6.10 Operator Precedence 21
3.7. Declarations 21
3.8. Functions 21
3.8.1 Function declaration 22
3.8.1.1 Function Calls 22
3.8.1.2 Return 22
3.9. Flow Control 23
3.9.1. Conditional statement 23
3.9.2. For statements 23
3.9.2.1 For 23
3.9.2.2 foreach element in array 24
3.9.2.3 foreach line in textfile, delimiter 24
3.10. Scope rules 24
3.10.1 Lexical scope 24
3.10.2 Global Scope 25
3.10.3 Statement Blocking Scope 25
3.10.2 Namespace 25
Chapter 4 Project Plan 27
4.1 Processes and Planning 27
4.2 Specification and Development 27
4.3. Software 27

4.3.1 Antlr 27
4.3.2 Java 28
4.4 Development Environment 28
4.5 Testing 28
4.6 Team Responsibilities 28
4.7 Project Timeline 28
4.8 Project Log 29
5.1 Components 30
5.2 Diagram 30
5.3 Interfaces 30
Chapter 6 Test Plan 32
6.1 Test Scripts 32
Chapter 7 Lessons Learned 36
7.1 Most Important Learning 36
7.2 Advice to PLT Students 36
Chapter 8 Appendix 38
8.1 Lexer 38
8.2 Parser 39
8.3 Walker 42

Table of Figures

Figure 1 HTML output for sample program....................10
Figure 2 HTML Report parts....................15
Figure 3 DFile project timeline....................29
Figure 4 Architectural Design....................30

Chapter – 1 Introduction

1.1 Definition

DFile is a new language that will facilitate HTML report creation from any delimited text file. DFile's ease of use, with its rich language, will allow non-programmers and programmers alike to create robust customized reports.

DFile has a convenient Java-like language style with several built-in functions allowing for easy text file manipulation. DFile also provides programmers with HTML rich set of attributes for styling and transforming reports.

1.2 Background

Users utilize cumbersome applications to produce reports from delimited text file - Microsoft Access, Crystal Reports, Report Writer to name a few, requiring programming experience to produce a favorable output. Using these applications users must first "convert" the text files, sometimes via a Wizard, prior to creating SQL like queries to filter the data for subsequent report "programming".

The regular user does not keep important data in expensive RDBMS (Oracle, MS SQL Server, Sybase), instead many maintain data in delimited text files. Besides their high retail price, RDBMS requires programming experience to properly maintain and manage data, making this a difficult storage solution for most users. As such, regular users are forced to maintain their own data in multiple delimited text files.

1.3 Goal

DFile ease of use is targeted to both non-programmers and programmers. It will parse and filter a delimited text file via built-in commands. This data manipulation of delimited text files, along with the rich set of HTML report styling and layout, allow for generation of reports without the need to modify or convert existing files.

The DFile Skunk team had two goal when undertaking the task of creating a new language – ease of use and portability.

1.4 Ease of Use

DFile's syntax models after very well know languages C/C++, Javascript, and C#, making it familiar to programmers.

Besides its robust set of operators, support for functions, foreach looping, and statement block scoping, DFile allows for HTML embedding within the code.

The familiar syntax coupled with the HTML embedding, allow DFile to produce reports with ease.

1.5 Portability

Since Dfile is interpreted and executed on a Java Virtual Machine, it may run on several platforms including PC and Unix based systems.

Chapter 2 - Language Tutorial

2.1 Getting Started

DFile has a short but powerful syntax. DFile uses the programming format similar to higher end languages as Javascript, C/C++, and C#. Each statement is terminated by semi-colon (;). DFile supports include files, global and local variable scoping, function definitions with return statements, for and foreach loops, and embedding of HTML code. See section 3 for detailed description of supporting language features.

2.2 Compiling a program.

DFile can be written using any text editor. By convention the program file should have an extension of 'df', but in the absence of a 'db' extension the compiler will take a file with any or no extension.

To compile a DFile do the following:

Java –jar <DFile.jar path> < <file path> > <output file>

<DFile.jar path> is the path where the DFile compiler resides.

<file path> is the path of the program file.

<output file> is output file for print statements. This is an optional argument, and if not given it will print to Standard Output.

Example: (with Dfiler residing in current directory)
Java –jar Dfiler.jar < example.df > ThisReport.html

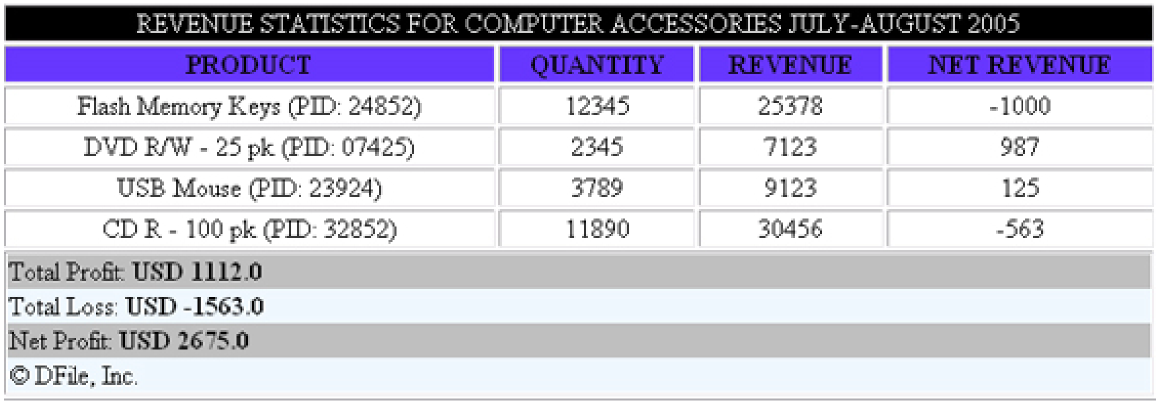
Here example.df will be compiled and the output will be placed in the ThisReport.html file.

2.3 Writing a Program

DFile can be written using any text editor. This program creates an HTML report from a comma separated file called *sampledata*.

```
// Include code from another DFile program.
include "reportsLib.df";

// Set the source for the data
source = "sampledata"; // note the lack of extension

// Set delimiting file character – comma delimited file
// Global scoping declared variable.
delim = ",";


// function to calculate profit
function whereProfit()
{
      // Return the number 3 field in file if value
      // is greater than 0
            return = (#3 > 0);
}
// function to calculate loss
function whereLoss()
{
      // Return the number 3 field value in file if it
      // is greater than 0
      return = (#3 < 0);
}

// Function to sum fields accepting 2 parameters
function sumFields(where, colNum)
{
      // local scope variable declaration
      sum = 0;
      // Loop through each column, as specified by parameter
      // colNum

      foreach line in source,delim
      {
            // conditional statement
            if (where())
            {
                  // sum the value
                  sum ~= sum + #colNum;
            }
      }

      return = sum;
}

// function to calculate total profit
function totalProfit()
{
      // return value from sumFields() function
      return = sumFields(whereProfit, 3);
}

function totalLoss()
{
      // return value from sumFields function
      return = sumFields(whereLoss, 3);
}

function footer()
{
      // local scoped variables declaration. Values being set by
      // function calls
      profit = totalProfit();
      loss   = totalLoss();
      net    = profit - loss;

      // HTML code
      bgcol1 = " bgcolor=""#C0C0C0""";
      bgcol2 = " bgcolor=""#F0F8FF""";

      // HTML code mixed with DFile code
      output = ""
      & "<table width=""100%"" border=""0"" cellspacing=""0"">"
      & "<tr><td" & bgcol1 & ">Total Profit: <b>USD " & profit &
"</b></td></tr>"
      & "<tr><td" & bgcol2 & ">Total Loss: <b>USD " & loss &
"</b></td></tr>"
      & "<tr><td" & bgcol1 & ">Net Profit: <b>USD " & net &
"</b></td></tr>"
      & "<tr><td" & bgcol2 & ">&copy; DFile, Inc.</td></tr>"
      & "</table>";

      return = output;
}

// Table title
heading = "REVENUE STATISTICS FOR COMPUTER ACCESSORIES JULY-AUGUST
2005";

// Table column header
headers = ["product", "quantity", "revenue", "net revenue"];

myfooter = footer();

// Generate report
generateReport(showAllLines, source, delim, heading, headers,
myfooter);
```

This will generate the following report:



REVENUE STATISTICS FOR COMPUTER ACCESSORIES JULY-AUGUST 2005

PRODUCT	QUANTITY	REVENUE	NET REVENUE
Flash Memory Keys (PID: 24852)	12345	25378	-1000
DVD R/W - 25 pk (PID: 07425)	2345	7123	987
USB Mouse (PID: 23924)	3789	9123	125
CD R - 100 pk (PID: 32852)	11890	30456	-563

Total Profit: **USD 1112.0**
Total Loss: **USD -1563.0**
Net Profit: **USD 2675.0**
© DFile, Inc.

Figure 1 HTML output for sample program

Chapter 3 - Language Reference Manual

3.1 Introduction

DFile is a new computer language that introduces the notion of a delimited text file as an object for data manipulation. DFile may be used on any computer using the Java Virtual Machine. As DFile is text-based, DFile programs may be written using any text editor such as emacs, vi, or Microsoft Notepad.

3.2. Syntax notation

For this manual, any italic notation signifies keywords. Examples are signified by Courier New font.

3.3. Lexical conventions

DFile's grammar consists of combined tokens to form expressions and statements. DFile uses the following tokens: keywords, identifiers, strings, and expression operators. Spaces, tabs, comment tags, and newlines are ignored, but they may be used to separate tokens.

3.3.1 Comments

The characters /* introduces a comment and it is terminated with the */ characters. The compiler will ignore all text between the start and ending comment characters. This type of comment style may span several lines.

Another form of commenting in Dfile is via the usage of // (double forward slash). Any text following these characters will be ignored up until the newline. As such, this commenting does not allow for multiple lines.

Both comment format will be familiar to C/C++ and C# programmers.
Example:

```
/* The DFile compiler will ignore this line
   and this line as well
*/

// The Dfile compiler will ignore this line
```

3.3.2 Identifiers (Names)

An identifier is a sequence of letters, digits, and underscores ("_"). The first character must be a letter. DFile is case sensitive with regard to identifiers; *Bob = 1 is not equivalent to bOB = 1.*

3.3.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

IF	*ELSE*	*INCLUDE*
FOREACH	*TRUE*	*FUNCTION*
PRINT	*FALSE*	*RETURN*
HEADING	*ISNUM*	*DELIMITER*
ISDEFINED	*COLNAMES*	*FOOTER*

3.3.4 Constants

DFile has two types of constants: number and string. Numbers can be either integers or floats. The number is further internally identified by DFile as a decimal (float).

3.3.4.1 Number constants

A number constant is a sequence of digits without a decimal point.
Example:
1232
2
12
92134123

3.3.4.1.1 Decimal constant

DFile has a second number datatype called the decimal. A decimal constant consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. The integer part is mandatory. The fraction part is optional. If there is no decimal point, the number will be treated as an integer. DFile's compiler will internally distinguish between an integer number and a decimal number by the presence of a decimal point.
Example:
12.123
34552.009

3.3.4.2 String constants

A string constant is one or more ASCII characters enclosed within double quotes: ‘‘"’’. Example:

"DFile is easy to use."

3.4 Statements

Program lines are composed of statements. Statements are executed in sequence. Each statement in turn is made of expressions. Each statement line, except where noted later, ends in a semi-colon (;). Exceptions to semi-colon terminating lines are in comments, function definitions, if and else definitions, and the for loop definitions. This rule applies to the definitions only and not to the body of the function, if..else, and for loop definitions. For examples, please see the appropriate sections below.

3.4.1 Expressions

Expressions are of one of the following forms:

identifier
constant
String
(expression)

3.4.2 Identifier

See section *3.3.2. Identifiers (Names)*

3.4.3 Constant

See section *3.3.4. Constants*

3.4.4 String

See section *3.3.4. Constants*

3.4.5 (expressions)

An expression enclosed within a left parenthesis and right parenthesis is also an expression. The programmer so as to override default precedence rules or just to provide clarity to other programmers who might read the code uses expressions of this form. Example:

```
        (3 + 4)
```

3.5. Keywords

DFile introduces several keywords.

3.5.1 If….Else

See section *3.9.1. Conditional statement*

3.5.1.1 Heading

The heading is the Title of the Table on the report.

See figure 2.

3.5.1.2 Footer

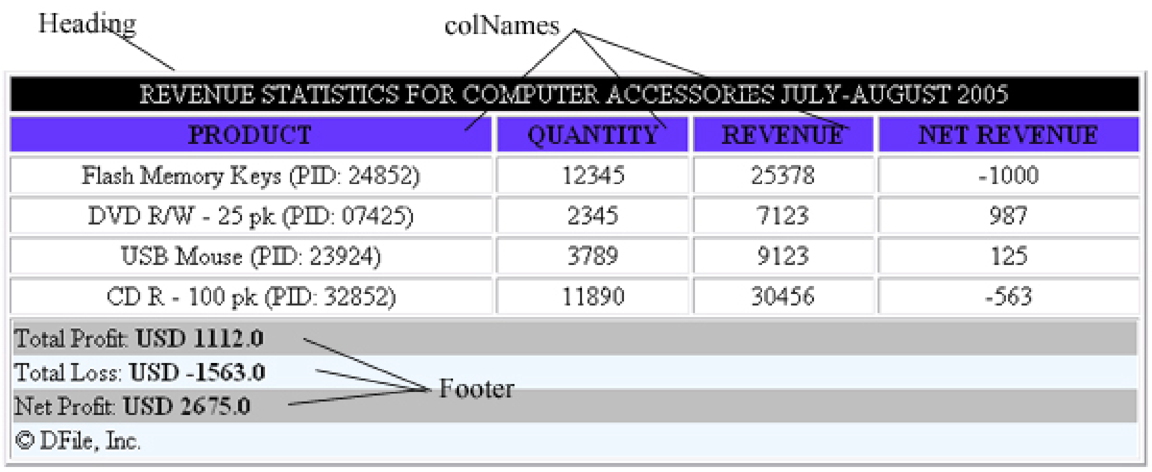
The footer is a separate row in the report table. It can be used to show any computational data.

See figure 2.

3.5.1.3 ColNames

ColNames is the heading for the table columns in the report.

See figure 2.



REVENUE STATISTICS FOR COMPUTER ACCESSORIES JULY-AUGUST 2005			
PRODUCT	QUANTITY	REVENUE	NET REVENUE
Flash Memory Keys (PID: 24852)	12345	25378	-1000
DVD R/W - 25 pk (PID: 07425)	2345	7123	987
USB Mouse (PID: 23924)	3789	9123	125
CD R - 100 pk (PID: 32852)	11890	30456	-563

Total Profit: **USD 1112.0**
Total Loss: **USD -1563.0**
Net Profit: **USD 2675.0**
© DFile, Inc.

Figure 2 HTML Report parts

3.5.1.2 For

3.5.1.3 ForEach

See section *3.9.2. For statement*

3.5.1.4 Delimiter

The delimiter specifies the character used for delimiting the source text file. If none is specified, the comma (,) is the default value. The delimiter attribute is used in the following form:

delimiter = *statement*;

Example:

```
delimiter = ":";
```

3.5.1.5 Include

The include statement is used to include other program code. This is very much as the C/C++ preprocessor directive *#include*. It is of the form:

include <file name>;

Example:

```
include "reportsLib.df"; // Include other functions written
```

3.5.1.6 Print

Calling print will specified argument to Standard Output. It has the form of

print(argument);

Where argument is of the form:

argument : *expression*

Example: to output a report in HTML format:

```
      output = ""
      & "<table width=""100%"" border=""0"" cellspacing=""0"">"
      & "<tr><td" & bgcol1 & ">Total Profit: <b>USD " & profit
      & "</b></td></tr>"
      & "<tr><td" & bgcol2 & ">Total Loss: <b>USD " & loss \
      & "</b></td></tr>"
      & "<tr><td" & bgcol1 & ">Net Profit: <b>USD " & net
      & "</b></td></tr>"
      & "<tr><td" & bgcol2 & ">&copy; DFile, Inc.</td></tr>"
      & "</table>";

      // this will print the table and associated, rows, and columns.
      print(output);
```

3.5.1.7 Function

See *3.8. Functions*

3.5.1.8 Return

See *3.8.1.2 Return*

3.5.1.9 isNum

This function will check the argument for a number. If the argument is a number then true is returned. Otherwise, false is returned.

IsNum has the following form:

IsNum number;

Example:

```
// Is x a number?
If (isNum x)
{
      print(“It is a number”);
}
else
{
      print(“It is NOT a number”);
}
```

3.5.1.10 isDefined

This function will check the argument to see if it has been defined.
If the argument has been defined then True is returned. Otherwise, false is returned.
IsDefined has the following form:

isDefined statement;

Example:

```
If (isDefined MyVariable)
{
      print(“It was defined previously”);
}
else
{
      print(“It has not been defined”);
}
```

3.6. Operators

DFile’s operators include the logical, relational, multiplicative, additive, equality, assignment, and the concatenation operators.

The significant number of operators present in DFile evolved in an attempt to maintain simplicity.

3.6.1. Logical Operators

The logical operators are || (logical or), && (logical and), and ! (negation).

3.6.1.1 expression || expression

The || operator returns true if either of its operands is true, and false otherwise.

3.6.1.2 expression && expression

The && operator returns true if both its operands are true, false otherwise.

3.6.2 Relational operators

The relational operators < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to) group left to right.

3.6.2.1 expression < expression, expression <= expression

3.6.2.2 expression > expression, expression >= expression

The relational operators < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to) all yield false if the specified relation is false and true if it is true.

3.6.3 Equality operators

The equality operators ==, and != group from left to right. The equality operators are analogous to the relational operators except for their lower precedence.

3.6.3.1 expression == expression

3.6.3.2 expression != expression

Both yield false if the specified relation is false and true if it is true.

3.6.4 Multiplicative

The multiplicative operators * (multiplication) and / (division) group left to right. Internally DFile will convert all numbers to decimals (floats), so result is not truncated.

3.6.4.1 expression * expression

The binary * operator indicates multiplication. Both expressions must be of type number.

3.6.4.2 expression / expression

The binary / operator indicates division. The same type considerations specified above for multiplication apply for division as well.

3.6.5 Additive operators

The additive operators + (addition) and - (minus) group left to right.

3.6.5.1 expression + expression

The result is the sum of the expressions where the expressions are both numbers. No other type of combination is allowed.

3.6.5.2 expression – expression

The result is the difference of the operands where the expressions are both numbers. No other type of combination is allowed.

3.6.6. Assignment

The assignment operator is = (equal), += (plus equal), -= (minus equal), *= (multiplication equal), /= (division equal), and the &= (concatenation equal) require an lvalue for assignment. These operators group right to left.

3.6.6.1. lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. Since DFile only has strings and numbers as data types, lvalue may be assigned by either type.

3.6.6.2 = Operator

It requires an lvalue as the left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

3.6.6.3 expression += expression

3.6.6.4 expression -= expression

3.6.6.5 expression *= expression

3.6.6.6 expression /= expression

3.6.6.7 expression &= expression

These operators shortcut two operations where the lvalue is assigned the value of the right side.

Example:

```
X += 1;
```

Is the same as:

```
X = X + 1;
```

3.6.7 Concatenation

The concatenation operator is the & (ampersand). It requires at least two strings to be added.

It has the form:

string expression & string expression

Example:

```
Var1 = “Hello”;
VarSpace = “ “;
Var2 = “World”;

Print(Var1 & VarSpace & Var2); // prints Hello World
```

3.6.8 # (Pound)

The # (pound) operator is used to refer to the column from the data source. It is zero indexed so #3 refers to the 4th column in the file.

It has the form:

#<column number>;

Example:

```
VarColumn = #3; //This retrieves the value of column 4 from the source.
```

3.6.9 ~= (Tilda equal)

This operator allows to access variable in the recent most outer scope.

Example:

```
// Car is a global variable
Car = “Toyota”;
// statement blocks create new scoping region.
{
      // Car is local to statement blocks delineated by {}
      Car = “Hummer”;
}

//This will print the global value
print Car; // prints “Toyota”
```

```
{
      /* new scoping area, but usage of ~= refers to global variable
      one level above the statement block
      */

      Car ~= “Hummer”;
}

print Car; //this prints “Hummer”
```

3.6.10 Operator Precedence

Operator precedence is (from higher to lower):
() {}
!
* /
+ - &
< > == !=
&& ||
,

3.7. Declarations

Declarations specify the interpretation given to each identifier. If the declaration reserves storage space, then it is called a definition. The data type of an identifier is inferred from its first usage. The programmer does not explicitly specify data types using keywords such as int or float.

3.7.1. Type specifiers

Type specifiers are number and string, which are implied and which cause the compiler to allocate the necessary memory for each specifier. Internally, DFile manipulates all numbers as decimals (floats).

As an example, the declarations
MyNumber = 1234;
MyOtherNumber = 123.34;
MyString = “MyFile.txt”;
declare a number (integer), a number (decimal), and a string.

3.8. Functions

Functions use a postfix expression followed by parentheses containing either an empty list or a comma-separated argument list. This argument list constitutes the arguments to the function. The argument becomes a parameter once the body of the function is being executed.

3.8.1 Function declaration

Functions are declared as follows:

function identifier (argument_expression_list) function_body;

3.8.1.1 Function Calls

Function calls are of the following form:

identifier (optional) = identifier(argument list);

Example:

```
Mysum =  AddNumbers(5, 8);
```

3.8.1.2 Return

A function may include a return statement. This assigns the value to the specifier that called the function.

It has the form:

return = (expression) ;

3.8.1.3 Construction

Functions are of the following form:

```
function declaration
{
        Function body
                Statements;
        Return = (expression);
}
```

Example:

```
Function Sum(a, b)
{
      if (isNum a && isNum b)
      {
            return = a + b;
      }
      else
      {
            return = 0;
      }
}
```

3.9. Flow Control

The two forms for program flow control are the if..else statement and the for loops. The for loop is futher supports foreach on arrays and on text files.

3.9.1. Conditional statement

The two forms of the conditional statement are
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
In both cases, the expression is evaluated and, if it is nonzero, the first sub-statement is executed. In the second case the second sub statement is executed if the expression is 0. As usual, the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered if.

Example:

```
if (x == 1)
{
	Statement;
}
else
{
	Statement;
}
```

3.9.2. For statements

The For loops has 3 modes of operation – a simple looping with number of iterations via its argument, a foreach element in array, and a foreach line in textfile, delimiter.

3.9.2.1 For

The for statement has the form

for (number of iterations)
{
Statement;
}

Example:

// this will increment the y five times

```
for(5)
{
      y = y + 4;
}
```

3.9.2.2 foreach element in array

The foreach has the following form:

foreach <element> in <array>

Example:

```
//field is an element of the array MyArray
foreach field in MyArray
{
      output &= "<td align=""center"">" & field & "</td>";
}
```

3.9.2.3 foreach line in textfile, delimiter

This foreach has the form:

foreaech <line> in <filename> , <delimiter>

Example:

```
// line is a field in the textfile filename delimited by delim.
foreach line in filename, delim
{
      if (shouldShowLine())
      {
            thisNumFields = 0;
      }
}
```

3.10. Scope rules

place the stuff about the symbol table and activation records here.

Lexical scope of an identifier is essentially the region of a program during which it may be used without drawing ‘‘undefined identifier’’ diagnostics.

DFile supports two scopes - global and local/statement block scope.

3.10.1 Lexical scope

The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function. It is an error to re-declare identifiers already declared within the body of the program. Since we do not allow nested functions, there are thus two lexical scopes: 1) the body of the program; and 2) the body of a function. These two scopes are separate; consequently, they cannot access each other's identifiers (unless by usage of the ~= operator. See 3.6.9)

3.10.2 Global Scope

This are variables declared outside of any statement blocks – identified by "{" and "}".

```
// Global scope
MyName = "Leroy";

{
      // not global scope, but not the same as first MyName.
      MyName = "Brown";
}

Print(MyName);// will print Leroy
```

3.10.3 Statement Blocking Scope

These are variables declared inside any statement blocks - identified by "{" and "}". Such variables have a lifetime only within the declared statement blocks. Each time a statement block is created then a new activation records is also created. Each activation record consists of symbol table and pointer to preceding activation record.

```
// Global scope
MyName = "Leroy";

{
      // not global scope, but not the same as first MyName.
      MyName = "Brown";
      Print(MyName); /// prints Brown
}

Print(MyName);// will print Leroy
```

3.10.2 Namespace

DFile will not reconcile namespace collisions – only one namespace is allowed. This means that an error will be raised if an identifier is declared more than once within the same lexical scope regardless of the identifier's data type or whether it is the identifier for a function.

3.10.4 Evaluation Order

DFile uses Applicative order evaluation. This means it evaluates function call arguments when they are called. As a consequence, DFile used shallow binding

Chapter 4 Project Plan

4.1 Processes and Planning

During this 3 month long project, the DFile team has meet on a regular basis at least once a week. Besides the regularly scheduled meetings, communication was also achieved via email and telephone. During the meetings we discussed and planed the project scope and definition, progress report for design, coding, testing, and documentation.

4.2 Specification and Development

Due to the limited project timeline and resources – not being able to work 40 hrs a week on this project alone, the DFile group implicitly decided to use a form of XP as the development methodology. The four members were quickly tasked with different parts of the project – design, coding, testing, and documentation. For some members the task boundaries overlapped.

Following XP practice, the task selections were made according to the strengths of each member. The white paper was the only documentation required – the Language Reference Manual came after parts of the code had been completed, and it was also , modified after all coding was done, allowing for quick coding and design revisions. The group worked without a team leader. We all acted as project leaders knowing what had to be done and when.

For the coding, much of it was done practicing pair programming – two developers sit side-by-side while coding.

4.3. Software

DFile was coded using Antlr and Java.

4.3.1 Antlr

Antlr facilitated coding the compiler for DFile. Antlr was selected due to its simple rule based syntax coupled with it's ability to work well with Java.

Antlr was used to create the Parser. The '//' was used for commenting code. Tokens, reserved words, are all in uppercase. When given multiple choices, the rules are placed in separate lines for clarity.

4.3.2 Java

The Java 1.5 syntax was used for the DFile development. Variables and objects are named interchanging from lower to upper camel case.

Embedded code within functions, loops, or statements is tabbed spaced for clarity – any code between '{' and '}'.

For comments, both '/*' with '*/' and '//' was used. Where required, a try-catch block was used to capture errors.

4.4 Development Environment

The major portion of the coding for Antlr was done in a Windows XP laptop. The code was then compiled and ran, via remote login, on the Clic Unix computers. See sections 4.3.1 Antlr and 4.3.2 Java.

4.5 Testing

We performed unit as well as system testing during our development. Also, we wrote several program scripts in DFile languge to ensure program . During sytstem testing, a script was created for regression testing. For unit testing, the same principle was applied – a new DFile script was created to test features.

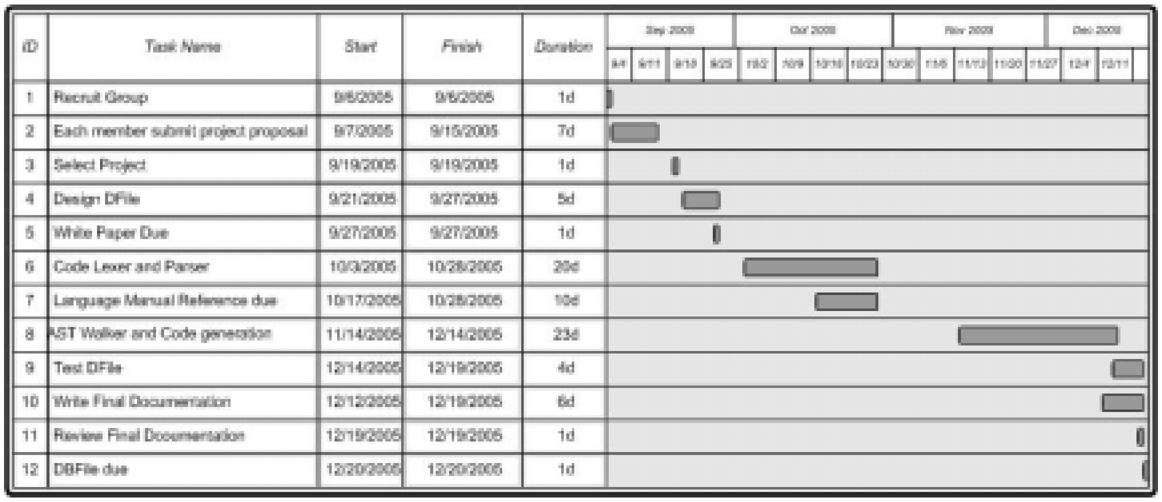
4.6 Team Responsibilities

The DFile task were divided among the four team members.

Member	Responsibility
Amrita Rajagopal	Architect, Senior Developer
Anton Ushakov	Quality Assurance
Erwin Polio	Documentation
Howie Vegter	Senior Developer

4.7 Project Timeline

The following is the Gant chart showing DFile timeline.



ID	Task Name	Start	Finish	Duration
1	Recruit Group	9/6/2005	9/6/2005	1d
2	Each member submit project proposal	9/7/2005	9/15/2005	7d
3	Select Project	9/19/2005	9/19/2005	1d
4	Design DFile	9/21/2005	9/27/2005	5d
5	White Paper Due	9/27/2005	9/27/2005	1d
6	Code Lexer and Parser	10/3/2005	10/28/2005	20d
7	Language Manual Reference due	10/17/2005	10/28/2005	10d
8	AST Walker and Code generation	11/14/2005	12/14/2005	23d
9	Test DFile	12/14/2005	12/19/2005	4d
10	Write Final Documentation	12/12/2005	12/19/2005	6d
11	Review Final Documentation	12/19/2005	12/19/2005	1d
12	DBFile due	12/20/2005	12/20/2005	1d

Figure 3 DFile project timeline

4.8 Project Log

Since much of the work was done in pair-programming, the need for VSS or CVS was unnecessary for version control. Thus DFile did not use any source safe tool.

Chapter 5 Architectural Design

5.1 Components

DFile consist of a 3 major components: Lexer, Parser, and Walker.

The job of the Lexer is to read input (command line or file) line by line, and to break it up into TOKENS. This was coded by Amrita

The Parser then takes the tokens and checks for syntax errors. It then converts the tokens into an AST. This was coded by Amrita and Howie.

The Walker *walks* the AST executing and evaluating statements based on TOKENS (from the Lexer). It also calls functions as required. This was coded by Amrita and Howie

5.2 Diagram

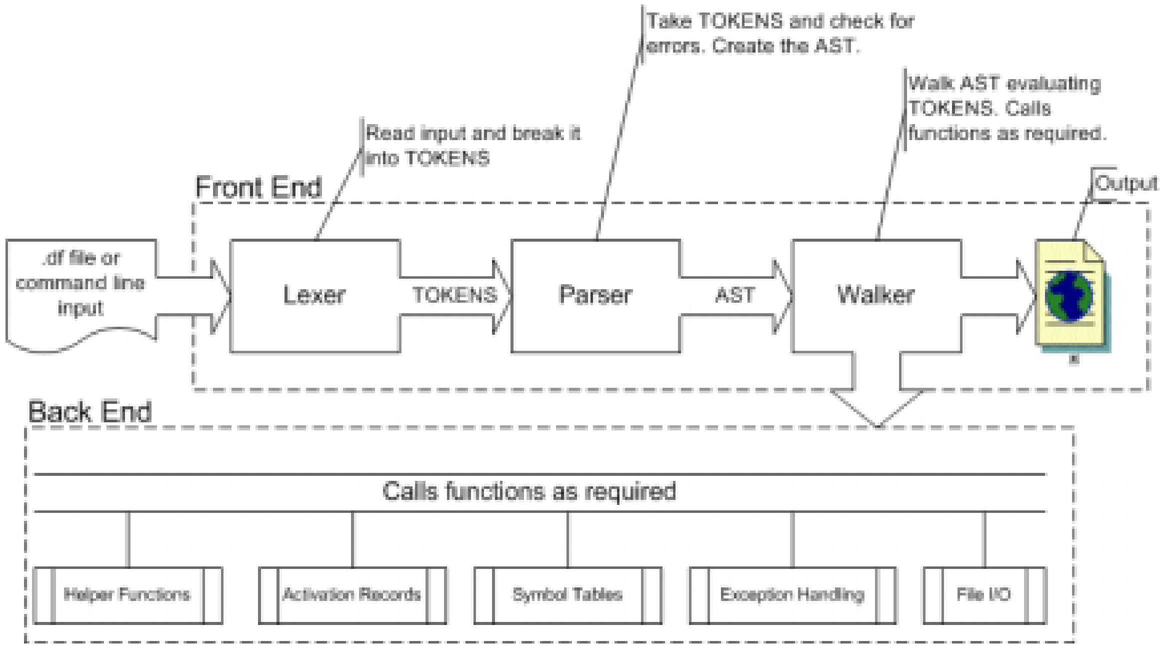

Take TOKENS and check for errors. Create the AST.
Walk AST evaluating TOKENS. Calls functions as required.
Read input and break it into TOKENS
Front End
Output
.df file or command line input
Lexer
TOKENS
Parser
AST
Walker
Back End
Calls functions as required
Helper Functions
Activation Records
Symbol Tables
Exception Handling
File I/O

Figure 4 Architectural Design

5.3 Interfaces

There are two main constructs in the walker: statement, expr (expression). Every line of input in considered a statement. Statements are of type print, include, assignment etc. (refer to parser). A statement can be an expression in which case a second step of evaluation is carried out by calling on the expr construct.

Various helper functions were written to help with processing the statements. No external modules were written. All the Java code is embedded within the Walker. There are two private classes which are defined and used within the walker: funcBody, and ActivationRecord.

```
  private class FuncObject
  {
        String[] args;
        AST body;

        private FuncObject(String[] args, AST body)
        {
                this.args = args;
                this.body = body;
        }
  }

  private class ActivationRecord
  {
        public ActivationRecord parent;
        public Hashtable slt; // symbol lookup table

        private ActivationRecord()
        {
                parent = activationRecords.empty() ? null :
activationRecords.peek();
                slt = new Hashtable();
        }
  }
```

Chapter 6 Test Plan

6.1 Test Scripts

```
#!/bin/sh

if [ "$1" =  '' ]
then
      echo "Usage: runtests <directory>"
      exit
fi

if [ -d $1 ]
then
      echo
else
      echo "runtests: $1 is not a directory"
      exit
fi

fail=0
pass=0
total=0

for file in $1/*.df
do
        total=`expr ${total} + 1`
        echo
        echo ========= $file =========
        echo

        name=`echo $file | sed -e 's/\.df//'`

        java -jar DFiler.jar < $file > $name.out 2>&1

        result=`diff $name.exp $name.out`


        if [ "$result" =  '' ]
        then
                pass=`expr ${pass} + 1`

                echo '  Pass'
        else
                fail=`expr ${fail} + 1`

                echo '  Failed'
                echo
                echo Code:
            echo --------------
            cat $file
            echo --------------
            echo Returned:
                echo

                cat ${name}.out

                echo
                echo Expected:
                echo
                cat ${name}.exp
        fi

        echo
        echo

      unlink $name.out

done
echo "Total number of tests: $total"
echo "Pass: $pass"
echo "Fail: $fail"
```

Testing scoping

```
y = 0;
for(5)
{
        y += 4;
        print y;
}
print y;

y = 0;
for(5)
{
        y = y + 4;
        print y;
}
print y;

y = 3;
for(5)
{
        y ~= y + 4;
        print y;
}
print y;
```

Testing function parameters:

```
function fun1(arg1)
{
    return = (arg1 + 2);
}

x = 40;
```

```
// should print 42
print fun1(x);
```

Testing function pointer:

```
function fun1()
{
    print "in fun1";
}

function fun2()
{
    print "in fun2";
}

function fun3(fun)
{
    fun();
}

// should print "in fun2"
fun3(fun2);

// should print "in fun1"
fun3(fun1);
```

Testing relational operators:

```
print (1 > 0);
print (0 > 1);
print (1 > 1);
print (1 < 0);
print (0 < 1);
print (1 < 1);
print (1 == 0);
print (1 == 1);
print (1 != 0);
print (1 != 1);

x = 1;
y = 0;

print (x > y);
print (y > x);
print (x > x);
print (x < y);
print (y < x);
print (x < x);
print (x == y);
print (x == x);
print (x != y);
print (x != x);


print (1 >= 0);
print (0 >= 1);
print (1 >= 1);
print (1 <= 0);
print (0 <= 1);
print (1 <= 1);
```

Testing return values:

```
function fun1()
{
   // return is a special keyword.  whatever value return has when the
function returns
   // is the value that the function will take on.
    return = (40 + 2);
}

// should print 42
print fun1();
```

Countless other test scripts were written and performed to Quality Assured DFile.

Chapter 7 Lessons Learned

7.1 Most Important Learning

We have a very good understanding what goes on each time a program is compiled.

7.2 Advice to PLT Students

- Writing a compiler in a 3 month time period, with many classes in between, is a daunting task. The task becomes the more difficult in the face of lacking technical documentation, but the short time period does not allow for such a formal and full project life cycle. As such, use development methodologies geared toward this approach. To succeed you must use any of the XP Programming practices – it is essential.

- As with any project, first seek to fully understand the complete scope, knowing that as time goes by the scope will change. Do your gathering requirement early and thoroughly, so as to allow for a better project planning.

- Select your partners carefully, seeking to complement the group's resources.

- Assign/elect a project leader in the beginning stage. It is imperative for one person to keep a check on the project timeline and maintain communication with the TA/professor. This resolves any possible confusion and makes sure that the team is on the same page and on schedule.

- Old project sources are very helpful in discovering hacks which may or may not be documented (for Antlr). So definitely spend some time in looking at older projects to analyze their architecture and syntax before starting on your own.

- Pay attention in class. We actually used ALL the grammar and compiler concepts explained in class e.g Activation Records and Symbol Tables, and it was a thrill to implement them. We resolved various issues by referring to the lecture notes.

- this is moreso an aid for future students than a lesson learned; but it would be helpful if there were code on the PLT website demonstrating error handling that prints the line and column number where the error occurred. this was more of a pain to handle than it should have been, especially since it was difficult to find useful info about this online.

- Keeping the documentation and product specifications synchronized isessential for testing. And vice versa, code testing forces the documentation to be updated and accurate.

- It is possible to get the walker done in about 2 weeks of very diligent, serious work. I think I told you before that it took 3 or 4 weeks.

Chapter 8 Appendix

8.1 Lexer

```
// The DFile Grammar
// Authors: Amrita Rajagopal, Howie Vegter

///////////////////////THE LEXER///////////////////////////

class DFileLexer extends Lexer;
options {
   testLiterals = false; // By default, don't check tokens against keywords
   k = 2;                // Need to decide when strings literals end
   charVocabulary = '\3'..'\377'; // Accept all eight-bit ASCII characters
   exportVocab = DFile;
}


PLUS     : '+' ;
MINUS              : '-' ;
TIMES  : '*' ;
DIV      : '/' ;
LESS     : '<' ;
GREAT : '>' ;
LESSEQ             : "<=" ;
GREATEQ            : ">=" ;
PLUSEQ  : "+=" ;
MINUSEQ : "-=" ;
MULTEQ  : "*=" ;
DIVEQ   : "/=" ;
CONCATEQ: "&=" ;
ASSIGN             : '=' ;
SEMI     : ';' ;
EQUALITY: "==";
NOTEQ : "!=" ;
LPAREN             : '(' ;
RPAREN             : ')' ;
LBRACE             : '{' ;
RBRACE             : '}' ;
LBOX    : '[' ;
RBOX    : ']' ;
AND      : "&&" ;
OR       : "||" ;
NOT      : '!';
COLON : ':';
COMMA              : ',';
CONCAT  : '&';
POUND   : '#';
TILDA  : "~=";

protected DIGIT : '0'..'9' ; // Any character between 0 and 9
protected LETTER : ('a'..'z' | 'A'..'Z' | '_');

ID options { testLiterals = true; }
: LETTER (LETTER|DIGIT)* ;

NUMBER : (DIGIT)+ ('.' (DIGIT)*)?;          // One or more digits

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

WS     : (' ' | '\t')+          { $setType(Token.SKIP); }
       ;

NL     : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
         { $setType(Token.SKIP); newline(); }
       ;

COMMENT : ( "/*" (
                options {greedy=false;} :
                (NL)
                | ~( '\n' | '\r' )
              )* "*/"
          | "//" (~( '\n' | '\r' ))* (NL)
         )                { $setType(Token.SKIP); }
       ;
```

8.2 Parser

```
//////////////////////THE PARSER//////////////////////////

class DFileParser extends Parser;
options {
    buildAST = true; // Enable AST building
    k = 2;          // Need to distinguish between ID by itself and ID ASSIGN
    exportVocab = DFile;
}
tokens {
  FUNCCALL;
  NEGATIVE;
  STATEMENT;
  NOTNODE;
  STMTBLOCK;
  ARGS;
  ARRAY_LIST;
  ARRAY_ELEM;
  COLINDEX;
  FOREACHINARRAY;
  FOREACHINFILE;
  FUNCDEF;
  FOR;
  INCLUDE;
  ISDEFINED;
  ISNUM;
  IF;
}


file: (statement)* EOF!
    ;

assignment_stmt:  ID ASSIGN^ (arr_expr | expr)
                        | ID ( (PLUSEQ^ | MINUSEQ^ | MULTEQ^ | DIVEQ^ | CONCATEQ^) expr)
                        ;

outer_assign: ID TILDA^ expr
                        ;

arr_expr: LBOX! array_list RBOX!
        ;

array_list: expr (COMMA! expr)*
                        { #array_list = #([ARRAY_LIST, "ArrayList"], array_list); }
            ;

expr: logical_expr ( OR^ logical_expr )* ;

logical_expr : relat_expr ( AND^ relat_expr)* ;

relat_expr  : arith_expr ( (LESS^ | GREAT^ | EQUALITY^ | NOTEQ^ | LESSEQ^ | GREATEQ^)
arith_expr)* ;

arith_expr : expr2 ( (PLUS^ | MINUS^ | CONCAT^) expr2 )*;

expr2 : expr3 ( (TIMES^ | DIV^) expr3 )* ;

expr3
    : (ID LPAREN) => func_call
    | ID
    | LPAREN! expr RPAREN!
    | MINUS! expr3
            { #expr3 = #([NEGATIVE, "NegativeNumber"], expr3); }
    | NOT! expr3
            { #expr3 = #([NOTNODE,"LogicalNot"], expr3); }
    | NUMBER
    | STRING
    | "true"
    | "false"
    | POUND! expr3
            { #expr3 = #([COLINDEX,"ColIndex"], expr3); }
    | (ID LBOX) => ID LBOX! (NUMBER | ID) RBOX!
            { #expr3 = #([ARRAY_ELEM,"ArrayElement"], expr3); }
    | "isDefined"! expr3
              {#expr3 = #([ISDEFINED, "IsDefined"], expr3); }
    | "isNum"! expr3
              {#expr3 = #([ISNUM, "IsNum"], expr3); }
    ;

statement : print_stmt SEMI!
              | include_stmt SEMI!
              | for_stmt
```

```
                | if_else
                | outer_assign SEMI!
                | assignment_stmt SEMI!
                | expr SEMI!
                | func_def
                | for_each
                | statement_block
                ;

statement_block:  LBRACE! (statement)* RBRACE!
                {#statement_block = #([STMTBLOCK, "StmtBlock"], statement_block); }
                    ;

print_stmt: "print"^ expr;

include_stmt: "include"! expr
                {#include_stmt = #([INCLUDE, "Include"], include_stmt); }
              ;

for_stmt : "for"! LPAREN! arith_expr RPAREN! statement_block
                {#for_stmt = #([FOR, "ForStatement"], for_stmt); }
        ;

if_else : "if"! LPAREN! expr RPAREN! statement_block
                (options {greedy = true;}: "else"! statement)?
                {#if_else = #([IF, "IfElse"], if_else); }
              ;

func_def
              : "function"! ID LPAREN! var_list RPAREN! statement_block
                {#func_def = #([FUNCDEF, "FunctionDefinition"], func_def); }
              ;

var_list : (ID ( COMMA! ID )*)?
                {#var_list = #([ARGS, "Args"], var_list); }
              ;

func_call: ID LPAREN! expr_list RPAREN!
                { #func_call = #([FUNCCALL, "FunctionCall"], func_call); }
              ;

expr_list
        : (expr ( COMMA! expr )*)?
              ;

for_each: ("foreach" ID "in" expr3 COMMA) =>
                        "foreach"! ID "in"! expr3 COMMA! expr3 statement_block
                        { #for_each = #([FOREACHINFILE, "ForEachInFile"], for_each); }
              | "foreach"! ID "in"! expr3 statement_block
                        { #for_each = #([FOREACHINARRAY, "ForEachInArray"], for_each); }
              ;
```

8.3 Walker

```
///////////////////////THE WALKER////////////////////////////

{

/*
 * Simple front-end for an ANTLR lexer/parser that dumps the AST
 * textually and displays it graphically.  Good for debugging AST
 * construction rules.
 *
 * Behrooz Badii, Miguel Maldonado, and Stephen A. Edwards
 */

import antlr.Token;
import java.io.*;
import antlr.CommonAST;
import antlr.CommonASTWithHiddenTokens;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
import java.util.*;
import java.lang.Math;
}
//class DFileTreeWalker extends TreeParser;
class DFiler extends TreeParser;

{
  Stack<ActivationRecord> activationRecords = new Stack<ActivationRecord>();
  int currentLineNumber    = 0;
  String data[][] = null;

  // Hashtable vars = new Hashtable();
  int funcCallDepth = 0; // keep track of whether we're at the top level (not within any function calls)

/*
  // copied from http://www.jguru.com/faq/view.jsp?EID=62654
  // it's actually in a separate file called CommonASTWithLines.java
  class CommonASTWithLines extends CommonAST
  {
     public CommonASTWithLines()
     { super(); }

     public CommonASTWithLines(Token tok)
     { super(tok); }

     private int line = 0;
     private int column = 0;
     public void initialize(Token tok)
     {
         super.initialize(tok);
         line=tok.getLine();
         column=tok.getColumn();
     }
     public int getLine()
```

```
    {
        return line;
    }
    public int getColumn()
    {
        return column;
    }
  }
*/

  private class FuncObject
  {
            String[] args;
            CommonASTWithLines body;

            private FuncObject(String[] args, AST body)
            {
                        this.args = args;
                        this.body = (CommonASTWithLines)body;
            }
  }

  private class ActivationRecord
  {
            public ActivationRecord parent;
            public Hashtable slt; // symbol lookup table

            private ActivationRecord()
            {
                        parent = activationRecords.empty() ? null : activationRecords.peek();
                        slt = new Hashtable();
            }
  }

  private ActivationRecord getTopAR(AST astAtErr)
  {
            if (activationRecords.empty())
            {
                        System.err.println(errPosPrefix(astAtErr) + "there is a programming error.  there are no
activation records.  we suck.  this should never happen, because we don't suck.");
                        System.exit(1);
            }

            return activationRecords.peek();
  }

  private void addAR(ActivationRecord arToAdd)
  {
            activationRecords.push(arToAdd);
  }


  private void addVarToSlt(AST astAtErr, String varname, Object val)
  {
            ActivationRecord ar = getTopAR(astAtErr);
            ar.slt.put(varname, val);
```

```
}

private Object getVarFromSlt(AST astAtErr, String varname)
{
        Object obj = getVarFromSltNull(astAtErr, varname);

        if (obj != null)
        {
                return obj;
        }
        else
        {
                // the variable was not found in any activation record
                System.err.println(errPosPrefix(astAtErr) + "unrecognized: " + varname + ". you suck");
                System.exit(1);
        }

        /** this code is unreachable, but javac complained about a lack of a return statement */
        return null;
}

/**
 * returns the variable named varname in the topmost AR
 * in which it is defined or null if it's not defined
 * in any AR.
 */
private Object getVarFromSltNull(AST astAtErr, String varname)
{
    ActivationRecord ar = getTopAR(astAtErr);
    Object ret = null;

    while (ar != null)
    {
         ret = ar.slt.get(varname);

         if (ret != null)
         {
              return ret;
         }

         ar = ar.parent;
    }

        // the var was not found.  return null
        return null;
}

private void replaceVarDef(AST astAtErr, String varname, Object val)
{
    ActivationRecord ar = getTopAR(astAtErr);
    Object var = null;

    while (ar != null)
    {
         var = ar.slt.get(varname);
```

```
            if (var != null)
            {
                            // the var was found in this activation record.  replace its val
                            ar.slt.put(varname, val);
                return;
            }

            ar = ar.parent;
        }

        // the var was not found.
        System.err.println(errPosPrefix(astAtErr) + "a variable named '" + varname + "' was not defined,
so you could not replace its value.  you tried to set it to '" + val.toString() + "'.  you suck.");
        System.exit(1);
  }

  /**
   * returns true if a function named varname is defined.  false otherwise.
   */
  private boolean funcNamedXDefn(AST astAtErr, String varname)
  {
        Object obj = getVarFromSltNull(astAtErr, varname);

        if (obj == null)     // a var named varname is not defined
                return false;

        if (obj instanceof FuncObject)
                return true;
        else
                return false;
  }

  private void verifyVarIsAssignable(AST astAtErr, String s, Object a)
  {
        if ( s.equals("return") && (funcCallDepth == 0) )       // cannot do "return = xx" unless inside a
function defn
        {
                System.err.println(errPosPrefix(astAtErr) + "return is a reserved word.  it can only be
used within a function definition.  and by the way, you suck.\n");
                System.exit(1);
        }
        if (funcNamedXDefn(astAtErr, s)) // returns true if a function named s is defined
        {
                System.err.println(errPosPrefix(astAtErr) + "a function named '" + s + "' is defined.  you
tried to define '" + s + "' to be '" + a.toString() + "'. you can't clobber defined functions.  you suck.");
                System.exit(1);
        }
  }

  private void verifyNotNull(AST astAtErr, String s, Object val)
  {
        if (val == null)
        {
                System.err.println(errPosPrefix(astAtErr) + "var '" + s + "' to overwrite was not found in
activation records.  you suck.");
                System.exit(1);
        }
  }

  /**
   *  this expression should take on the value of whatever the variable "return" was set to.
   *  if "return" wasn't set, the expression will return null, which is ok.
   */
  private Object getReturn(AST astAtErr)
  {
        ActivationRecord ar = getTopAR(astAtErr);
        return ar.slt.get("return");
  }

  /**
   * filename is the name of the file we're walking.  used for error messages.
   * error prefixes will look like this "filename.dg:20:29: "
   */
  public void setFileName(String filename)
  {
        this.fileWeAreWalking = filename;
  }

  /**
   * if filename isn't specified, then we'll assume we're walking stdin.
   * if we're walking stdin, then we don't include a filename in the error prefix:
   * error prefixes will look like this "20:29: "
   */

  private String fileWeAreWalking;

  public static void main(String args[])
  {
    try
    {
      DataInputStream input = new DataInputStream(System.in);

      // Create the lexer and parser and feed them the input
      DFileLexer lexer = new DFileLexer(input);
      DFileParser parser = new DFileParser(lexer);

      parser.setASTNodeClass("CommonASTWithLines"); // this is from a CommonASTWithLines.java file
in the same directory.
      parser.file(); // "file" is the main rule in the parser

      // Get the AST from the parser
      CommonASTWithLines parseTree = (CommonASTWithLines)parser.getAST();
//      CommonAST parseTree = (CommonAST)parser.getAST();

      // Print the AST in a human-readable format
//      System.out.println(parseTree.toStringList());

      // Open a window in which the AST is displayed graphically
//      ASTFrame frame = new ASTFrame("AST from the DFile parser", parseTree);
//      frame.setVisible(true);

        // the walker
        DFiler texasRanger = new DFiler();
            texasRanger.setFileName(null); // represents stdin as opposed to an included file
        texasRanger.dfile(parseTree);
    }
    catch(StackOverflowError e)
    {
            System.out.println("\n\nNon-terminating condition. Probably a non-terminating recursive
function. you really suck. \n");
            System.exit(1);
    }
    catch(NullPointerException n)
    {
            System.out.println("The source file was empty. Nothing to do. you suck. ");
            n.printStackTrace();
            System.err.println(n.toString());
    }
    catch(Exception e)
    {
            System.err.println("Exception: "+e);
    }
  }

  public int getInt (AST astAtErr, Object o)
  {
        int result = 0;
        try
        {
                    if (o instanceof String)
                    {
                            result = Integer.parseInt((String)o);
                    }
                    else
                    {
                    result = ((Double)o).intValue();
                    }
        }
        catch(ClassCastException e)
        {
             System.err.println(errPosPrefix(astAtErr) + "Cannot perform this operation on non-integers. you
suck.");
             System.exit(1);
        }
        return result;
  }

  public Object[] getArray (AST astAtErr, Object o)
  {
        Object[] result = null;
        try
        {
             result = (Object[])o;
        }
        catch(ClassCastException e)
        {
             System.err.println(errPosPrefix(astAtErr) + "Cannot perform this operation on non-arrays. you
suck.");
```

```
            System.exit(1);
        }
        return result;
    }

    public double getDouble (AST astAtErr, Object o)
    {
            double result = 0.0;
            try
            {
                    if (o instanceof String)
                    {
                            result = Double.parseDouble((String)o);
                    }
                    else
                    {
                            result = ((Double)o).doubleValue();
                    }
            }
            catch(NumberFormatException e)
            {
                    System.err.println(errPosPrefix(astAtErr) + "Cannot perform this operation on non-
numbers.  you suck. (expected a number; found '" + o.toString() + "' which is a " +
o.getClass().getCanonicalName() + ")");
                    System.exit(1);
            }
            catch(ClassCastException e)
            {
                    System.err.println(errPosPrefix(astAtErr) + "Cannot perform this operation on non-
numbers. you suck.");
                    System.exit(1);
            }
            return result;
    }

    public String getString (AST astAtErr, Object o)
    {
        String result = "";
        try
        {
                    if (o instanceof String)
                    {
                            result = (String)o;
                    }
                    else if (o instanceof Double)
                    {
                            result = ((Double)o).toString();
                    }
                    else if (o instanceof Boolean)
                    {
                            result = ((Boolean)o).toString();
                    }
                    else
                    {
                            throw new ClassCastException();
                    }
        }
        catch(ClassCastException e)
        {
                System.err.println(errPosPrefix(astAtErr) + "Can only perform this operation on strings,
booleans, and numbers. you suck.");
                System.exit(1);
        }

      return result;
  }

  public boolean getBoolean (AST astAtErr, Object o)
  {
            boolean result = false;
        try
        {
                result = ((Boolean)o).booleanValue();
        }
        catch(ClassCastException e)
        {
                System.err.println(errPosPrefix(astAtErr) + "Cannot perform this operation on non-boolean. you
suck.");
                        System.exit(1);
        }
            catch (Exception e)
            {
                        System.err.println(errPosPrefix(astAtErr) + e.toString());
                        e.printStackTrace();
                        System.exit(1);
            }
            return result;
  }

  private String errPosPrefix(AST astAtError)
  {
            String filename = (this.fileWeAreWalking == null) ? "" : fileWeAreWalking + ":";

            if (astAtError == null)
                        return filename + "Unknown error location: ";

            return filename + astAtError.getLine() + ":" + astAtError.getColumn() + ": ";
  }
}

dfile
{
  if (activationRecords.empty())
  {
            ActivationRecord mainAR = new ActivationRecord();
            activationRecords.push(mainAR);
  }
}
: ( statement )*
;

statement
```

```
{ Object a, ignore, ob1, ob2, ob3; double d1, d2; String s; String[] args; boolean b; }
: #("print" a=expr)
        {
                System.out.println(a);
        }
| #(INCLUDE a=expr)
        {
                if (!(a instanceof String))
                {
                        System.err.println("include filename must be a string.  you suck.");
                        System.exit(1);
                }

                String includeFile = (String)a;

                try
                {
                InputStream input = (InputStream) new FileInputStream(includeFile);


                        DFileLexer lexer = new DFileLexer(input);
                        DFileParser parser = new DFileParser(lexer);

                        parser.setASTNodeClass("CommonASTWithLines");
                        parser.file(); // "file" is the main rule in the parser

                        // Get the AST from the parser
                        CommonASTWithLines parseTree = (CommonASTWithLines)parser.getAST();
                        //      CommonAST parseTree = (CommonAST)parser.getAST();

                        // Print the AST in a human-readable format
//                      System.out.println("included AST: " + parseTree.toStringList());

                        DFiler walker = new DFiler();
                        walker.setFileName(includeFile);

                        // share the symbol table
                        walker.addAR(this.getTopAR(#INCLUDE));

                        walker.dfile(parseTree);
                }
                catch (FileNotFoundException e)
                {
                System.err.println(errPosPrefix(#INCLUDE) + e.toString() + "\ninclude of file '" +
includeFile + "' failed.  file does not exist.  furthermore, you suck.");
                        System.exit(1);
                }
                catch ( Exception e )
        {
                System.err.println(errPosPrefix(#INCLUDE) + e.toString() + "\ninclude of file '" +
includeFile + "' failed for an unknown reason.  somebody sucks.  I'm not sure if it's you or me.");
                        System.exit(1);
        }
        }
| #(ASSIGN ID a=expr)
        {
```

```
                        s=#ID.getText();

                        verifyVarIsAssignable(#ASSIGN, s, a);

                        addVarToSlt(#ASSIGN, s, a);
            }
| #(PLUSEQ ID a=expr)
            {
                        s=#ID.getText();

                        verifyVarIsAssignable(#PLUSEQ, s, a);

                        Object val = getVarFromSltNull(#PLUSEQ, s);

                        verifyNotNull(#PLUSEQ, s, val);

                        replaceVarDef(#PLUSEQ, s, getDouble(#PLUSEQ, val) + getDouble(#PLUSEQ, a));
            }
| #(MINUSEQ ID a=expr)
            {
                        s=#ID.getText();

                        verifyVarIsAssignable(#MINUSEQ, s, a);

                        Object val = getVarFromSltNull(#MINUSEQ, s);

                        verifyNotNull(#MINUSEQ, s, val);

                        replaceVarDef(#MINUSEQ, s, getDouble(#MINUSEQ, val) - getDouble(#MINUSEQ,
a));
            }
| #(MULTEQ ID a=expr)
            {
                        s=#ID.getText();

                        verifyVarIsAssignable(#MULTEQ, s, a);

                        Object val = getVarFromSltNull(#MULTEQ, s);

                        verifyNotNull(#MULTEQ, s, val);

                        replaceVarDef(#MULTEQ, s, getDouble(#MULTEQ, val) * getDouble(#MULTEQ, a));
            }
| #(DIVEQ ID a=expr)
            {
                        s=#ID.getText();

                        verifyVarIsAssignable(#DIVEQ, s, a);

                        Object val = getVarFromSltNull(#DIVEQ, s);

                        verifyNotNull(#DIVEQ, s, val);

                        double aVal = getDouble(#DIVEQ, a);

                        if (aVal == 0)
```

```
                {
                        System.err.println(errPosPrefix(#DIVEQ) + "you tried to divide " + s + " by
zero.  you suck.");
                        System.exit(1);
                }

                replaceVarDef(#DIVEQ, s, getDouble(#DIVEQ, val) / aVal);
        }
| #(CONCATEQ ID a=expr)
        {
                s=#ID.getText();

                verifyVarIsAssignable(#CONCATEQ, s, a);

                Object val = getVarFromSltNull(#CONCATEQ, s);

                verifyNotNull(#CONCATEQ, s, val);

                replaceVarDef(#CONCATEQ, s, getString(#CONCATEQ, val) +
getString(#CONCATEQ, a));
        }
| #(TILDA ID a=expr)
        {
                String varname = #ID.getText();

            ActivationRecord ar = getTopAR(#TILDA);
        ar = ar.parent;

                boolean assigned = false;

        while (ar != null)
        {
                        if (!ar.slt.containsKey(varname))
                        {
                                ar = ar.parent;
                        }
                        else
                        {
                                ar.slt.put(varname,a);
                                assigned = true;
                                break;
                        }
            }

            if (!assigned)
         {
             System.err.println(errPosPrefix(#TILDA) + "var '" + varname + "'to overwrite was not
found in parent's symbol lookup table.  you suck.");
             System.exit(1);
         }
        }
| ignore=expr
| #(STMTBLOCK           {
                        ActivationRecord foreachAR = new ActivationRecord();
             activationRecords.push(foreachAR);
                }
```

```
                        (stmt:. { statement(#stmt);})*)
                {
                        Object testReturn = getReturn(#stmt);
                        if (testReturn != null)
                        {
                                ActivationRecord parent = getTopAR(#stmt).parent;
                                if (parent == null)
                                        System.err.println("return keyword defined in global scope.
you suck.");

                                parent.slt.put("return",testReturn);
                        }
                        activationRecords.pop();
                }
| #(IF a=expr thenblock:. (elseblock:.)?)
        {
                b = getBoolean(#IF, a);

                if (b)
                {
                        statement(#thenblock);
                }
                else if(elseblock != null)
                {
                        statement(#elseblock);
                }
        }
| #(FOR a=expr execblock:.) //for i=1:2 {}
        {
                d1 = getDouble(#FOR, a);

                for (int i = 0; i < d1; i++)
                {
                        statement(#execblock);
                }
        }
| #(FUNCDEF funcname:. args=getvars funcbody:.)
        {
                if (funcCallDepth != 0)      // disallow nested function defns.  must be at top level (not
within a function call)
                {
                        // cry
                        System.err.println(errPosPrefix(funcname) + "Nested function definitions are
not allowed.  Neither are people who suck.");
                        System.exit(1);
                }
                s = funcname.getText();
                addVarToSlt(#FUNCDEF, s, new FuncObject(args, funcbody));
        }

| #(FOREACHINFILE line:. ob2=expr ob3=expr forstmt:.)
        {
                String linename = line.getText();
                String source = getString(#FOREACHINFILE, ob2);
                String delimiter = getString(#FOREACHINFILE, ob3);
```

```
          try
          {
               BufferedReader br = new BufferedReader(new FileReader(source));

               Vector lines = new Vector();
               Vector lineVec;
               String lineStr;

               for (int i = 0; (lineStr = br.readLine()) !=null; i++)
               {
                    lineVec = new Vector();

                    StringTokenizer st = new StringTokenizer(lineStr, delimiter);

                    for (int j = 0; st.hasMoreElements(); j++)
                    {
                         lineVec.add(st.nextElement());
                    }

                    lines.add(lineVec.toArray(new String[0])); // returns the line as a String array
               }

               data = (String[][])lines.toArray(new String[0][0]);     // convert the vector to a String[][]
array


               for (int i = 0; i < data.length; i++)
               {
                                   currentLineNumber = i;

                  ActivationRecord foreachAR = new ActivationRecord();
                        activationRecords.push(foreachAR);

                                   addVarToSlt(#FOREACHINFILE, linename, data[i]);

                                   statement(#forstmt);

                                   activationRecords.pop();
               }
          }
          catch(IOException e)
          {
               System.out.println(e);
          }
         }

| #(FOREACHINARRAY ArrElemName:. ob2=expr forarrstmt:.)
         {
          String arrElemName = ArrElemName.getText();
          Object[] array = getArray(#FOREACHINARRAY, ob2);


                  for (Object elem : array)
                  {
                           ActivationRecord foreachArrAR = new ActivationRecord();
                       activationRecords.push(foreachArrAR);
```

```
                                addVarToSlt(#FOREACHINARRAY, arrElemName, elem);
                                statement(#forarrstmt);

                                activationRecords.pop();
                        }
                }
;

getvars returns [String[] args]
{ Vector var_list; args = null;}
: #(ARGS {var_list = new Vector();}
        (s:ID { var_list.add(s.getText()); })*

        //convert Vector to array of String
        {
                args = new String[var_list.size()];
                int i = 0;
                for (Enumeration e = var_list.elements() ; e.hasMoreElements(); )
                {
                        try
                        {
                                args[i] = (String)e.nextElement();
                        }
                        catch(ClassCastException x)
                        {
                                System.out.println(errPosPrefix(#ARGS) + "Function arguments must
be Strings only. you suck. ");
                                System.exit(1);
                        }

                        i++;
                }
        }
  )
;

expr returns [Object r]
{ Object a,b; double d1,d2; String s1,s2,s; Vector argvals = new Vector(); boolean b1,b2; r = null; }
: ID
        {
                s=#ID.getText();

                r = getVarFromSlt(#ID, s);
        }
| #(PLUS a=expr b=expr)
    {
                d1 = getDouble(#PLUS, a);
                d2 = getDouble(#PLUS, b);

         r = d1 + d2;
    }
| #(MINUS a=expr b=expr)
    {
         d1 = getDouble(#MINUS, a);
         d2 = getDouble(#MINUS, b);
```

```
                r = d1 - d2;
        }
    | #(TIMES a=expr b=expr)
        {
                d1 = getDouble(#TIMES, a);
                d2 = getDouble(#TIMES, b);

                r = d1 * d2;
        }
    | #(DIV a=expr b=expr)
        {
                d1 = getDouble(#DIV, a);
                d2 = getDouble(#DIV, b);

                if (d2 == 0)
                {
                        System.err.println(errPosPrefix(#DIV) + "cannot divide by zero.  you suck.");
                                System.exit(1);
                }

                r = d1 / d2;
        }
    | #(NEGATIVE a=expr)
        {
                d1 = getDouble(#NEGATIVE, a);

                r = -1 * d1;
        }
    | n:NUMBER
            {
                        try
                        {
                                r = Double.parseDouble(n.getText());
                        }
                        catch (NumberFormatException e)
                        {
                                System.err.println(errPosPrefix(n) + "expecting a number.  found '" + n + ".' you
    suck.");
                                System.exit(1);
                        }
            }
    | #(LESS a=expr b=expr)
            {
                        d1 = getDouble(#LESS, a);
                        d2 = getDouble(#LESS, b);

                        r = d1 < d2;
            }
    | #(GREAT a=expr b=expr)
            {
                        d1 = getDouble(#GREAT, a);
                        d2 = getDouble(#GREAT, b);

                        r = d1 > d2;
            }
```

```
| #(LESSEQ a=expr b=expr)
            {
                        d1 = getDouble(#LESSEQ, a);
                        d2 = getDouble(#LESSEQ, b);

                        r = d1 <= d2;
            }
| #(GREATEQ a=expr b=expr)
            {
                        d1 = getDouble(#GREATEQ, a);
                        d2 = getDouble(#GREATEQ, b);

                        r = d1 >= d2;
            }
| #(EQUALITY a=expr b=expr)
            {
                        r = a.equals(b);
            }
| #(NOTEQ a=expr b=expr)
            {
                        r = !(a.equals(b));
            }
| #(CONCAT a=expr b=expr)
            {
                        s1 = getString(#CONCAT, a);
                        s2 = getString(#CONCAT, b);

                        r = s1 + s2;
            }
| str:STRING
            {
                        r=str.getText();
            }
| #(AND a=expr b=expr)
       {
             b1 = getBoolean(#AND, a);
             b2 = getBoolean(#AND, b);

             r = b1 && b2;
       }
| #(OR a=expr b=expr)
       {
             b1 = getBoolean(#OR, a);
             b2 = getBoolean(#OR, b);

             r = b1 || b2;
       }
| #(NOTNODE a=expr)
       {
             b1 = getBoolean(#NOTNODE, a);

             r = !b1;
       }
| "true"
            {
                        r = true;
```

```
		}
	| "false"
		{
			r = false;
		}
	| #(ARRAY_LIST { Vector arr_vals = new Vector();} (array_list:.
		{
			arr_vals.add(expr(#array_list));
		})+)
		{
			r = arr_vals.toArray();
		}
	| #(ARRAY_ELEM a=expr b=expr)
		{
			Object[] arr = getArray(#ARRAY_ELEM, a);
			int index = getInt(#ARRAY_ELEM, b);

			try
			{
				r = arr[index];
			}
			catch (ArrayIndexOutOfBoundsException e)
			{
				System.err.println(errPosPrefix(#ARRAY_ELEM) + "array index out of
bounds.  you suck.");
				System.exit(1);
			}
		}
	| #(COLINDEX a=expr)
		{
			int colindex = getInt(#COLINDEX, a);
			r = data[currentLineNumber][colindex];
		}
	| #(FUNCCALL funcname:. (arguments:.
					{
				Object argvalue = expr(#arguments);

						argvals.add(argvalue);
					})*)
		{
			s = funcname.getText();

			funcCallDepth += 1;

			try
			{
				FuncObject fub = (FuncObject)getVarFromSlt(#FUNCCALL, s);

				String[] argsID = fub.args; //args within func declaration
				CommonASTWithLines funcbody = fub.body;

				if(argsID.length != argvals.size())
				{
					System.err.println(errPosPrefix(funcname) + "Wrong number of
arguments to function '" + s + "'. expected " + argsID.length + " args; received " + argvals.size() + " args.
you suck. ");
```

```
                                        System.exit(1);
                                }

                                ActivationRecord funcAR = new ActivationRecord();
                                activationRecords.push(funcAR);

                                int i =0;
                                for (Enumeration e = argvals.elements() ; e.hasMoreElements(); )
                {
                                        //System.out.println("arg: "+ e.nextElement());

                                        addVarToSlt(#FUNCCALL, argsID[i], e.nextElement());
                                        i++;
                                }

                                statement(#funcbody);

                                // this expression should take on the value of whatever the variable "return" was
set to.
                                // if "return" wasn't set, the expression will return null, which is ok.
                                r = getReturn(#FUNCCALL);

                                //remVarFromTopSlt("return");

                                activationRecords.pop();

                        }
                        catch (ClassCastException e)
                        {
                                System.err.println(errPosPrefix(funcname) + e.toString());
                                System.exit(1);
                        }

                        funcCallDepth -= 1;
                }
| #(ISNUM a=expr)
                {
                        r = true;

                try
                {
                        if (a instanceof String)
                        {
                            Double.parseDouble((String)a);
                        }
                        else
                        {
                        ((Double)a).doubleValue();
                        }
                }
                catch(NumberFormatException e)
                {
                                r = false;
                }
                catch(ClassCastException e)
                {
                                r = false;
                }
                }
| #(ISDEFINED def:.)
                {
                        String varname = def.getText();
                        r = getVarFromSltNull(#ISDEFINED, varname) != null;
                }
;
```

1189 lines of code total for the g file.

Walker – texas ranger
Compiler = Dfiler
Command language as standard input separated by semicolon
Extension is not required for compilation.