d-File Language Reference Manual

Erwin Polio
Amrita Rajagopal
Anton Ushakov
Howie Vegter

COMS 4115.001
Thursday, October 20, 2005
Fall 2005
Columbia University
New York, New York

Note: Much of the content and structure of this document is based on the C Reference Manual [1].

1. Introduction

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

2. Syntax notation

For this manual, any italic notation signifies examples. Keywords are signified by `Courier New` font.

3. Lexical conventions

d-File's grammar consists of combined tokens to form expressions and statements. d-File 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.

3.1 Comments

The characters // (double forward slash) introduce a comment. The comment is terminated with the newline character. Comments only span the length of strings from the start of the double forward slash to the end of the first found newline. To do multiple line comments, each line must be preceded by //. d-File will ignore all text after the double forward slash //. This comment form will be familiar to C/C++ and C# programmers.
Example:

// The d-File compiler will ignore this line

3.2 Identifiers (Names)

An identifier is a sequence of letters, digits, and underscores ("_"). The first character must be a capital letter; this helps the programmer distinguish between keywords (which are all lowercase) and identifiers (which must begin with a capital letter). d-File is case sensitive with regard to identifiers; *Bob = 1 is not equivalent to bOB = 1.*

3.3 Keywords

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

```
print          source
delimiter      where
for            if
function       else
number         string
datafile       column
return         footer
```

3.4 Constants

d-File has two types of constants: `number` and `string`. The number is further internally identified as an integer or a decimal.

3.4.1 Number constants

A `number` constant is a sequence of digits without a decimal point.
Example:

1232
2
12
92134123

3.4.1.1 Decimal constant

d-File 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. d-File'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

3.4.2 String constants

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

"My pants are on fire."

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.

4.0 Expressions

Expressions are of one of the following forms:

identifier
constant
String
(expression)

4.1 *identifier*

See section 3.2.

4.2 *constant*

See section 3.4.

4.3 *string*

See section 3.4.2.

4.4 (expressions)

An expression enclosed within a left parenthesis and right parenthesis is also an expression. Expressions of this form are used by the programmer to override default precedence rules or just to provide clarity to other programmers who might read the code.
Example:

(3 + 4)

5. Objects

d-File introduces a data storage object of type DataFile.

5.1 DataFile

A `datafile` is an object used for abstracting the delimited text file. The datafile consists of several attributes and methods: `source`, `where`, `delimiter`, `column`, `footer`, and `print`.

5.1.1 Source

The `datafile source` attribute is used to associate the delimited text file with the `datafile`. It will be the source of all the record manipulation. The source method is in the following form:

`source` = *statement*;

Example:

source = “c:\Accounting\GL\December-2004\Reconciliation.txt”;

5.1.2 Where

The `where` method is used to filter the records identified by the `source` method. The `where` clause is a statement consisting of `strings`, `numbers`, and operators (including the equality operand). The where clause is in the following form:

`where` = *statement*;

Example:

`where` = column(0) > 1000 && column (4) = “California”;

5.1.3 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` = `":";`

5.1.4 Column

The `column` attribute is for specifying a vertical set of tupels in the `datafile`. This identification is made by an index number relating to the delimited file where an index of 0 relates to the first column, an index of 1 relates to the second column, and so on. It is possible to index beyond the number of columns available, but no data will be returned. The `column` attribute is used in the following form:

`column`(*index);*

Example:

`column`(3) ;

5.1.5 Footer

The footer attribute is for specifying string constants to be printed at end of a report. It is used in the following form:

`footer`*(index) = statement;*

5.1.6 Print

Calling print will begin the report creation on the format specified in the argument. It has the form of

`print`*(argument);*

argument : “XML” or “HTML”

Example: to output a report in HTML format:

`print`*(“HTML”)*;

6. Operators

d-File’s operators include the logical, relational, multiplicative, additive, equality, and assignment operators.

6.1. Logical Operators

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

6.1.1 *expression && expression*

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

6.1.2 *expression || expression*

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

6.2 Relational operators

The relational operators < (less than), > (greater than), == (equal compare), and != (not equal compare) group left to right.

6.2.1 *expression < expression*
6.2.2 *expression > expression*

The operators < (less than), > (greater than), == (equal compare), and != (not equal) all yield “false” if the specified relation is false and “true” if it is true.

6.3 Equality operators

The equality operators ==, and != group from left to right. Both yield “false” if the specified relation is false and “true” if it is true.

6.3.1 *expression == expression*
6.3.2 *expression != expression*

The equality operators are analogous to the relational operators except for their lower precedence.

6.4 Multiplicative

The multiplicative operators * and / group left to right.

6.4.1 *expression * expression*

The binary * operator indicates multiplication. If both operands are `number` integers, the result is a `number` integer. If both are `number` decimals, the result is a `number` decimal. If one is a `number` integer and one is a `number` decimal, the result is a decimal. Both expressions must be of type `number`.

6.4.2 *expression / expression*

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

6.5 Additive operators

The additive operators + and – group left to right.

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.

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.

6.6. Assignment

The assignment operator is = (equal) and it groups right to left.
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.

6.6.1. *lvalue = expression*

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

6.7. Operator Precedence

Operator precedence is (from higher to lower):

```
() {}
!
* /
+ -
< > == !=
&& ||
,
```

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`.

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

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.

8.1 Function declaration

Functions are declared as follows:

```
    function identifier (argument_expression_list) function_body;
```

8.1.1 Function Calls

Function calls are of the following form:

```
    identifier (optional) = identifier(argument list);
```

Example:

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

8.1.2 Return

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

```
    return ;
    return ( expression ) ;
```

8.1.3 Construction

`Functions` are of the following form:

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

Example:

```
    function AddNumbers(number x, number y)
    {
        number tmp;
```

```
            tmp = x + y;
            return tmp;
      }
```

9. Flow Control

The two forms for program flow control are the `if..else` statement and the `for` loop.

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 substatement 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;
      }
```

9.2. For statement

The `for` statement has the form

```
for ( expression1 = range specifier)
      {
        Statement;
      }
```

Example:

```
      // this will increment the y five times
      for(x=5:10)
      {
            y = y + 4;
      }
```

9.3. Break statement

The statement `break ;` causes termination of the innermost enclosing `for` statement; control passes to the statement following the terminated statement.

10. Scope rules

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

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.

10.2 Namespace

d-File 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.

11. d-File Example

Here is a complete d-File program which reads a document, sets a delimiter, sets the source, and produces an XML report.

```
// Set location of file
source = "C:\RevenueStatsJulyAugust2005.txt";

// Set character delimited
delimeter =  “:”;

// Unless specified, the field names will be taken from the file header.
//Column(0) name is “Product”;
//Column(1) name is “Quantity”;
//Column(6) name is “Revenue”;
//Column(7) name is“Net Revenue”;

// Criteria for selecting data
where = column(1)  > 2000 && column(6) > 5000;

// Sum the revenue (Profit)
Profit = SumRevenue();

//Set footer using variables resturned from functions
footer(1)= "Total Profit: USD" + Profit

//Create report
print(“XML”);

Function SumRevenue()
{
        // Sum Revenue rows 0 through 30
        tmp = 0;
        for( i = 0:30)
        {
                // Only do positive
                if (column(7) > 0)
                {
                        tmp = tmp + column(7);
                }
        }
        return tmp;
}
```

REFERENCES

1. Kernighan, B. W, and Ritchie, D. M. ‘‘The C Programming Language.’’ Upper Saddle River, New Jersey, Prentice Hall Software Press, 1988, Appendix A.