DAVE

Language Reference Manual

Hyun Seung Hong hh2473

Min Woo Kim mk3351

Fan Yang fy2207

Chen Yu cy2415

October 26, 2015

Contents

1 Introduction

For the last few decades developers have created a variety of tools to help with data management and analysis. SQL and R language, for example, are two of the most prominent tools in this field, being widely utilized by business analysts, social scientists, statisticians and many more. Popular as they are, there still remain certain scenarios where both tools are imperfect. For instance, social scientists today often need to incorporate data of distinct formations from different sources to conduct their statistical analysis. SQL may work well for this job, yet it is designed for relational databases, a system not exactly suitable for most statistical tasks. R language, on the contrary, is fully optimized for statistical analysis with tons of functions and libraries, yet it still lacks the simplicity and usability of SQL in terms of managing datasets.

Data Analytics Visualization with Ease (DAVE) is a programming language optimized for data retrieval, manipulation, analysis and visualization. DAVE is designed with cross-dataset operations in mind, which is fairly common yet complicated to achieve with current toolsets in today's data-intensive tasks. The primary goal of DAVE is to allow the programmers to validate datasets, incorporate (parts of) datasets from different sources, split up oversized datasets, conduct statistical analysis and visualize critical data.

This document is a reference manual for the DAVE language, and it follows the organizational structure set by Dennis M. Ritchie in his *C Reference Manual*.

2 Lexical Conventions

2.1 Character Set

DAVE supports a subset of ASCII characters, whose decimal codes in the ASCII table range from 32 ([SPACE]) to 126 (~). In other words, DAVE understands [SPACE], !, " ,#, $, %, &, ', (,), *, +, -, ., /, number 0-9, :, ;, <, =, >, ?, @, upper-case letter A-Z, lower-case letter a-z, [, \,], ^, _, \, {, |, }, ~. Escape sequences, such as \t, \n, and \r, are also included in DAVE's character set.

2.2 Comment

A comment block starts with /* and ends with */. Multi-line comments and single-line comments are handled by the same rule. Comments are ignored by the syntax.

2.3 Identifiers

A legitimate identifier is a sequence of upper-case letters A-Z, lower-case letters a-z, numbers 0-9, and/or the underscore _. Additionally, the identifier must start with upper-case letters, lower-case letters or underscore. Identifiers are case-sensitive; “countryGDP” and “countrygdp” are two separate identifiers. Identifiers cannot equal to the reserved words (i.e., keywords), and each can be at most 31 characters.

2.4 Reserved keywords

The following identifiers are reserved; they cannot be used as ordinary identifiers. Also, their spellings must be exactly same as follows:

`int`	`float`	`bool`	`str`	`tbl`
`rec`	`fld`	`none`	`if`	`elif`
`else`	`for`	`while`	`break`	`continue`
`true`	`false`	`return`	`def`	`constant`

2.5 Constants

DAVE programmers can declare primitive constants in four data types (whose details would be specified in latter sections): *integer* (`int`), *float* (`float`), *string* (`str`), and *boolean* (`bool`).

Integer constants are decimal numbers consisting of a series of digits (0-9) and a leading sign character (+ or -). It is fine not to include the sign character; in this case the statement will always be parsed as a positive integer number.

Float constants are decimal numbers with a decimal separator (.), a series of digits (0-9) before the separator (integer part), a series of digits after the separator (fraction part) and a sign character (+ or -). Similar to *integer constants*, the sign character could be omitted, rendering the constant positive automatically. Additionally, the integer part and the fraction part cannot be absent at the same; a digit of 0 would be automatically filled into the absence if the user decide not to declare the integer part or fraction part.

String constants are a series of supported ASCII characters surrounded by a pair of quotation marks (“”). A backslash (\) is required for certain characters to be included in the string, such as:

Character	Input as	Name
\n	\n	Line Feed
\t	\t	Horizontal Tab

\r	\r	Carriage Return
”	\”	Quotation Mark
\	\\	Backslash

Boolean constants can have either the value *true* or the value *false*. The values are case-sensitive; inputs like *True*, *TRUE*, *False* or *FALSE* are not Boolean values.

3 Syntax notation

In this manual, syntactic categories are indicated by *italic* type, while literal words, characters and code examples are in written in `consolas`. Optional expressions are indicated by $_{opt}$.

4 Conversions

There are number of operators that can cause conversion between different data types.

4.1 Number conversions

`int` and `float` can be converted to each type by using `float()` and `int()` functions, respectively. When converting `float` to `int`, decimal values will be rounded down. For example:

```
int x = 1              /* x = 1 */
float y = float(x)     /* y = 1.0 */
int z = int(y + 2.5)   /* z = 3 */
```

4.2 String conversions

`int`, `float`, and `bool` can be converted to a string by using `str()` function. For example:

```
str age = str(11)                    /* age = ‘11’ */
str hello = ‘Im ’ + age + ‘ years old’ /* hello = ‘Im 11 years old’
*/
str is_that_true = str(true)         /* is_that_true = ‘true’ */
```

4.3 Boolean conversions

`int`, `float`, and `str` can be converted to a boolean value by using `bool()` function. x is converted to `false` if x is `none`, empty or omitted, otherwise it returns true. For `int` and `float`, `bool(x)` returns true for x values of 0 or 0.0 and false otherwise. For example:

```
bool e = bool(‘’)            /* e = false */
```

```
bool s = bool('0')          /* s = true */
bool z = bool(0.00)         /* z = false */
bool n = bool(none)         /* n = false */
```

4.4 fld conversions

`int[]`, `float[]`, `bool[]`, and `str[]` can be converted to a fld by using fld() function. A field name must be specified when converting. For example:

```
int[] num = [21, 23, 24, 26]
fld ages = fld(num, 'age')
```

`fld` can be converted back to `int[]`, `float[]`, `bool[]`, or `str[]` using `int()`, `float()`, `bool()`, and `str()`, respectively. For example:

```
fld ages = fld([21, 23, 24, 26], 'age')
int[] num = int(ages)       /* num = [21, 23, 24, 26] */
```

4.5 tbl conversions

`fld` and `rec` can be converted to a `tbl` using `tbl()` function. One or more `fld` or `rec` instances can be converted into one `tbl`. When converting multiple `fld`'s into a `tbl`, their sizes must be equal in order to have same number of values for each record. When converting multiple `rec`'s into a `tbl`, they can have different `fld` names, and missing `fld`'s will be considered `none`. For example:

```
fld a = fld([21, none, 24, 26], 'age')
fld b = fld(['James', none, 'Chen', 'Fan'], 'name')
tbl roster = tbl(a, b)      /* table with 4 records and 2 fields */
rec c = {name: 'Min Woo', age: 22}
rec d = {grade: 99, name: 'John'}
tbl roster2 = tbl(c, d)     /* table with 2 records and 3 fields */
```

5 Expressions

Expressions in DAVE are a combination of identifiers, constants, function calls and operators.

5.1 Primary Expressions

Primary expressions involving . and function calls group left-to-right.

5.1.1 *identifier*

An identifier is a name used to identify an entity in a program. It is preceded by its type (such as `str`), and it should be suitably declared as explained at above (§2.3).

5.1.2 *constant*

A decimal, floating constant, string, character, or boolean is a primary constant. A decimal is of type `int`, floating constant is of type `float`, string and character are of type `str`, and boolean is of type `bool`.

5.1.3 (*expression*)

A parenthesized expression is simply an expression surrounded by a parenthesis; it is equal to an unadorned expression.

5.1.4 *primary-expression* (*expression-list*$_{opt}$)

A function call is a primary expression followed by an optional list of expressions in parenthesis. The list of expressions may be empty, a single expression, or a comma-separated list of expressions. A copy is made of each actual parameter passed to the function, so changes to its formal parameters by a function does not affect the values in the actual parameters.

5.1.5 *primary-expression* [*expression*]

A primary expression followed by an expression in square brackets is a primary expression. A primary expression must be an array of a primitive type (`int[]`, `float[]`, `bool[]`, or `str[]`), `tbl` or `fld`, and an expression must be an integer within the range of the array size. An expression can also contain a colon (:), which has integers on the left and right. This specifies the range of primary expressions to return. When an integer is not specified on the right of the colon, the length of the array will be used as default. The resulting object is an array of primary expression. For example:

```
str[] names = [‘James’, ‘Min Woo’, ‘Chen’, ‘Fan’]
str first = names[0]   /* first = ‘James’ */
str[] rest = names[1:] /* rest = [‘Min Woo’, ‘Chen’, ‘Fan’] */
```

5.1.6 *object-type.member-of-object-type*

An object type followed by a dot followed by a member of object type is a primary expression. Element(s) of `rec` or `tbl` can be accessed by specifying the name of the `fld` in it. For example:

```
fld a = fld([21, none, 24, 26], ‘age’)
fld b = fld([‘James’, none, ‘Chen’, ‘Fan’], ‘name’)
tbl roster = tbl(a, b)       /* table with 4 records and 2 fields */
fld ages = roster.age        /* ages = a */
rec c = tbl[0]               /* c = {age: 21, name: ‘James’} */
str name = c.name            /* name = ‘James’ */
```

5.2 Unary Operators

All unary arithmetic operations have the same priority, and unary operators group right-to-left.

5.2.1 - *expression*

The unary - operator yields the negative of the expression. The type of expression must be `int` or `float`.

5.2.2 ! *expression*

The unary ! operator yields the logical negation of the expression. The type of expression must be `int` or `bool`. The result of the expression is 1 or true if the value of the expression is 0 or false, 0 or false if the value of the expression is non-zero or true.

5.2.3 ++ *expression*

The unary ++ operator increments the expression by 1. The type of expression must be `int`.

5.2.4 -- *expression*

The unary `--` operator decrements the expression by 1. The type of expression must be `int`.

5.2.5 *expression* ++

The result of the expression is the value of the object referred to by the expression. After the result is returned, the expression is incremented by 1.

5.2.6 *expression* --

The result of the expression is the value of the object referred to by the expression. After the result is returned, the expression is decremented by 1.

5.3 Multiplicative operators

multiplicative operators group left-to-right.

5.3.1 *expression* * *expression*

The binary * operator indicates multiplication. If both operands are `int`, the result is `int`; if one operand is `int` and the other is `float`, the result is `float`; if both operands are `float`, the result is `float`.

5.3.2 *expression* / *expression*

The binary / operator indicates division. The types of operands and results are the same as multiplication.

5.3.3 *expression* % *expression*

The binary % operator indicates remainder of the division of the expressions. Both operands must be `int`, and the result is `int`.

5.4 Additive operators

Additive operators group left-to-right.

5.4.1 *expression* + *expression*

The result is a sum of the two expressions. If both operands are `int`, the result is `int`; if one operand is `int` and the other is `float`, the result is `float`; if both operands are `float`, the result is `float`; if both operands are `str`, the result is `str`. No other type combinations are allowed.

5.4.2 *expression* - *expression*

The result is a difference of the two expressions. The types of operands and the results are the same as addition.

5.5 Comparison operators

Comparison operators <, >, ==, >=, <=, and != compare the values of two objects and yield either `true` or `false`. The operators have the same priority. Comparison operators group left-to-right.

When the compared objects are not of the same type, If they are numbers, they are converted to a common type. Otherwise, they always compare unequal on == or !=, and throw an error on any other comparison operator.

The objects of the same type are compared according to the following rules:

- Numbers are compared arithmetically.
- Strings are compared lexicographically based on their ASCII numeric equivalents.
- are compared lexicographically, each corresponding element pair at a time robustly.

5.6 Assignment operators

Assignment operators group right-to-left.

5.6.1 *expression = expression*

The value of the left expression is replaced by the expression on the right. The operands need to be of same type, and it can be either `int`, `float`, `str`, `bool`, `fld`, `tbl`, or `rec`.

5.6.2 *expression += expression*
5.6.3 *expression -= expression*
5.6.4 *expression *= expression*
5.6.5 *expression /= expression*
5.6.6 *expression %= expression*

The behavior of the expression of the form “E1 op= E2” is equivalent to “E1 = E1 op E2”.

6 Declarations

Before our usage of identifiers within the functions, we need to give a declaration in order to specify their interpretation. The declaration of identifier might contain the decl-specifier part and the declarator-list part.

As for the decl-specifier, it contains the type-specifier and sc-specifier (storage class). Unlike C and Java, the type-specifier of DAVE is compulsory while the sc-specifier could be omitted. And the declarator-list could be just an identifier, or several composing a list.

```
declaration:
  type-specifier sc-specifier_opt  declarator-list_opt
```

6.1 Storage class declarations

The DAVE language demands three levels of storage class, the `auto`, the `static` and the `extern`. These three kinds differ from their reserved amount of storage.

The extern class would be further explained in the Scoping part of this manual. If the user have not covered the sc-specifier in his declaration, it would be automatically set as `auto`. As for the static class, the storage would be initialized as zero or none depends on its type specifier if the user neglected.

```
sc-specifier:
  auto
  static
  extern
```

6.2 Type Specifier

The type-specifiers in DAVE might cover the kinds of constants mentioned above. The declaration of rec is alike the declaration of struct in C language, also including `type-decl-list.` Thus would be further illustrated later.

```
type-specifier:
  int
  char
  float
  bool
  str
  tbl
  tbl {title-decl-list}
  fld
  rec
  rec {title-decl-list}
```

6.3 Declarators

The declarator-list could be a sequence of declarators separated with a single comma.

The type of storage class of objects could be known due to its corresponding specifier.

```
declarator:
  identifier
  declarator()
  declarator[ constant-expression_opt]
```

In the following, we might discuss more about the last two kinds of declarators.

If the declarator has the form like `D()`, it would be treated as the type “function returning …”, where the “...” is the type which the identifier have had if the declarator is simply D.

If the declarator has the form like `D[]`, or `D[constant-expression`$_{opt}$`]`, the constant expression should be an integer determining the storage distribution. Such a declarator makes the identifier have the type “array”. Also, the array could be extended from an existing array, from a single-axis array to a two-dimensional one (matrix). Generally, we seldom utilize more than three-dimensional array in our DAVE applications.

Not all the possible combine situations above would be workable, especially for the ones linked with `fld`, `tbl` and `rec`. Returning array in the function or constituting array with function is not permitted. But returning a `fld` or `rec` is somewhat accessible. Also, these three types would not support array declarator, the declaration of array is only accessible to the primitive type specifiers, regarding `int`, `char`, `float`, `bool`, `str`.

As an example, the declaration

```
int i, f(), a[], b[][];
```

declares an integer i, a function f returning an integer, an array of integer numbers, a two-dimensional array of integer numbers. Thus might be the same for the basic specifiers like char, float, double, str.

As for the remaining three, the declaration is a little complicated, thus also limited. The declaration

```
fld fld1 = fld([18, 21, 25, none], 'age')
```

declares a field fld1, the storage type might differ from the right side of the equation, the user might not need to declare that explicitly, the parser would recognize thus automatically. The informations before the comma are the data stored in the field and the information after the comma is the title of the field.

In this example, It indicates that the field1 contains four entities, while one of them is none, and the remaining three are integers. The field may be just analogous to an one-dimensional array, except the implicit type definition. However, the most distinct difference between field and one-dimensional array is that it need to have a title, otherwise the corresponding upper layer tbl could not recognize it correctly. The conversion between one-dimensional array and filed could be found in the Conversion Chapter.

To be remembered, the `tbl`, `rec`, `fld` types do not support array extension. If you want to keep track of two fields synchronously, utilize the conversion from `fld` type to `tbl` type as introduced in part four. But function returning field type would be accepted.

```
fld f()
```

6.4 Record and Table declarations

Above, we have introduced the declaration of primitive types and `fld` type, however the declaration of `rec` and `tbl` would be more like the `struct` declaration in C language.

Since the data stored in the record possess different types of attribute, we need to point out the title of each entity, referring which field they would link with. The difference between our record and the traditional struct is that we do not need to indicate the type of the data, they might be recognize by the parser, however the title of the data is indispensable.

The declaration

```
rec rec1 = {name: 'Fan Yang', age: 22}
```

declares a record rec1, the data are separated with commas. And the part before the colon is the title of data. The function returning record would be accepted in DAVE. If you want to keep track of two records synchronously, also you need to perform the conversion to table type to achieve thus.

Also, the above assignment is equalized to

```
rec rec1 {name, age} = {'Fan Yang', 22}
```

It this pattern, the title of the data are listed in the left, So the data in the right side need to match the left side correctly, not just number but also sequence. This might be not as flexible as the first one.

The declaration of table type need to be done through the conversion, not to burden too much on the parser. The conversion between the three types `tbl`, `rec` and `fld` could be referred

from the Conversion Chapter, which we would not duplicate here. Just gave an simple example of table declaration here.

The declaration

```
tbl tbl1 = tbl(fld([23,24],'age'), fld(['Fan Yang', 'James'], 'name'))
```

or

```
tbl tbl1 = tbl({name: 'Fan Yang', age: 23}, {name: 'James', age: 24})
```

or

```
tbl tbl1 {name, age} = tbl({'Fan Yang', 23}, {'James', 24})
```

could both declare and create the identical table, no matter from two fields or two records.

Thus, the record and table declarations both accepts two kinds of syntaxes as follows,

```
tbl
tbl {title-decl-list}
rec
rec {title-decl-list}
```

7 Simple Statements

Each simple statement is identified by a newline character at the end. Several simple statements may occur on a single line separated by semicolons.

7.1 Expression statements

Expression statements are used to evaluate and store a value, or to call a function. Any valid expressions that are comprised within a single logical line is a statement. Here are some examples:

```
5 + 2
10 > 3
x++; y = 2x + 5
```

7.2 Assignment statements

Assignment statements are used to bind or rebind references to values and to modify attributes of mutable objects. Here are some examples:

```
x = 2; x = 4
myObj.myAttr = "Changed"
```

7.3 `print` statement

`print` stringifies each expression and writes the resulting object to standard output. A newline character is written at the end if not specified otherwise. Multiple expressions can be written in a single line when concatenated by the + operator.

7.4 `save` statement

`save` writes a formatted text, which represents a DAVE table, to a specified path. For example:

```
str path = ‘documents/roaster.txt’
fld a = fld([21, 23, 24, 26], ‘age’)
fld b = fld([‘James’, ‘Min’, ‘Chen’, ‘Fan’], ‘name’)
tbl table = tbl(a, b)
str option = ‘w’        /* option for overwrite */
save(path, table, option)
```

7.5 `load` statement

`load` validates the given text file, converts it into the `tbl` object and returns the object. For example:

```
str path = ‘documents/roaster.txt’
str[] field_list = [‘age’, ‘name’]
/* list of field names to fetch (default all if not included) */
int[] record_list = [0:4]
/* list of record indices to fetch (default all if not included) */
tbl table = load(path, field_list, record_list)
```

7.6 `return` statement

`return` terminates the current function call with the expression list. For example:

```
return true
```

If the expression list is absent, none is substituted.

7.7 `break` statement

The break statement terminates the loop without executing the rest of the code block.

7.8 `continue` statement

The continue statement skips the rest of the code block and triggers the next iteration.

8 Compound Statements

Compound statements enclose any number of statements within one or more clauses, comprised within an indented code block that follows a colon. Compound statements usually span multiple lines.

8.1 `if` statement

The `if` statement conditionally executes a set of statements, based on the truth value of a given expression. The following is the general form of an `if` statement:

`if` (*condition1*):
    then-statement
`elif` (*condition2*):
    elif-statement
`else:`
    else-statement

If *condition1* evaluates to true, then *then-statement* is executed and *elif-statement* and *else-statement* are ignored. If *condition1* evaluates to false and *condition2* evaluates to true, *elif-statement* is executed and *then-statement* and *else-statement* are ignored. If neither *condition1* nor *condition2* is met, *else-statement* is executed.

8.2 `for` statement

The `for` statement is used to iterate over the elements of an iterable object and execute. The following is the general form of a `for` statement:

`for` *index* `in` *sequence*[*start*:*end*]:
    block of code

8.3 `while` statement

The `while` statement is used to execute a block of code as long as a specified condition is true. The following is the general form of a `while` statement:

`while` (*condition*):
    block of code

8.4 Function definitions

A user can define a function object with none to many abstract parameters. The execution of a function definition binds the function name to a function object. The function body is only executed when the function is called.
The following is the general form of a function definition:

`def` *func_name*(*parameters*):
    function body

9 Scope

DAVE delimits code blocks by indentation. A variable declared in a certain code block cannot be referenced outside the code block. Variables that are only referenced within a code block are implicitly global. Variables declared within a function body are local to the function body, regardless of its relative indentation level, unless explicitly declared as global.

10 File Format

A valid DAVE table text needs to begin with ‘<DAVE>’ and end with ‘</DAVE>’. The header includes the data type and name of each field. Each record is separated by a newline character, and each field within one record is followed by a semicolon.

For example, the following code:

```
str path = ‘documents/roaster.txt’
fld a = fld([21, 23, 24, 26], ‘age’)
fld b = fld([‘James’, ‘Min’, ‘Chen’, ‘Fan’], ‘name’)
tbl table = tbl(a, b)
str option = ‘w’        /* option for overwrite */
save(path, table, option)
```

Generates the following text file:

```
<DAVE>
str age; int name;
21; James;
23; Min;
24; Chen;
26; Fan;
</DAVE>
```

REFERENCES

1. Ritchie, Dennis M. *C Reference Manual*. Bell Laboratories, 1973.
2. "The Python Language Reference." *The Python Language Reference — Python 2.7.10 Documentation*. N.p., n.d. Web. 06 Aug. 2015.