ScriptEdit Language Reference Manual

Bhavesh Patira
bp2214@columbia.edu

Bethany M. Soule
bms2126@columbia.edu

Deni Pejanovic
dp2232@columbia.edu

Marc Vinyes
mv2258@columbia.edu

March 5, 2007

1 Introduction

ScriptEdit is a language that allows you to automatically generate text from a limited set of instructions. You can write new files with this text, or insert it into an existing file. The instructions may use the text of the edited file, external files or text generated by other standard input/output based applications as input. ScriptEdit is similar to a macro processor in the sense that replaces text with other text, but it can also create new files from one single source.

The main goal of this language is to allow the user to edit files and the script operations that are needed to create their content all from within one single source file. Often, editing content text files (HTML, LaTeX, XML, etc) is a process that involves several different steps and programs - like separate bash scripts, a text editor, and other console programs (e.g. using ImageMagick to edit images or using Matlab to create graphs that will be linked). ScriptEdit is a way to put all those different process calls together with the content text file.

2 Overall idea

A script-edit code is a text file that contains script-edit statements interleaved with text from another language. We will refer to the latter as string constants (a more detailed definition is given later). Statements process this text and may output text in the position where they are written.

3 Main lexical conventions

We have the following kinds of tokens: string constants, keywords, characters that separate the arguments of a keyword, identifiers, conditions, mathematical expressions, the declaration operator(`#`) and the content operator (`$`). Their definitions are provided throughout this document.

Whitespace, including tabs and newlines are generally relevant everywhere except when indicated otherwise within this manual. For example: leading and trailing whitespaces are not ignored:

{`My String Variable`} is not equivalent to {`My      String     Variable`} is not equivalent to {`    My String Variable    `}.

3.1 String constants

String constants are all consecutive sets of ascii characters including but not limited to dashed, numbers and spaces that aren't keywords, characters that separate the arguments of a statement, identifiers, conditions, mathematical expressions, the declaration operator(`#`) or the content operator (`$`).

The `#, $, (, ), {, }` characters can be escaped with backslash \ if needed.

3.2 Comments

Since ScriptEdit is intended to be embedded within the text of another file, we do not provide a ScriptEdit-specific comment. To comment your code use the native comment style in the string constants.

For example if you are editing an HTML file you would use:

```
<HTML>
        <!-- your comments -->
<HTML>
```

3.3 Identifiers (Names)

An identifier is a case sensitive sequence of letters, digits, and underscore (_). The first character cannot be a digit.

3.4 Keywords, declaration and content operators

The following identifiers are keywords that are reserved for specific use as follows: The `#` within a line signifies the start of a ScriptEdit command, and is usually immediately proceeded by a keyword without any space character seperation.

```
$file     #file     #def        #if
#else     #while    #exec       #calc
#for      #next     #do         #gettoken
#write    #break    #continue
```

The declaration (`#`) and content operators (`$`) may be considered a string constant or part of a variable/function declaration depending on rules that are specified by their semantics.

The lexical conventions of each statement are explained along with its semantics.

3.5 Conditions

Conditions are exclusively used as part of the `#if` and `#while` constructs. They evaluate to either true or false.

Allowable binary operators are:

```
=, <, >, >=, >=, !=
```

Conditions are composed by pairs of statements seperated by the binary operator where each statement evaluates to a string value for comparison. Since all variables are identified by their string value, only strings are compared.

Conditional statements can be compounded using the `&` or `|` operators. Parentheses can be used to maintain proper associativity within the conditional statement.

For example:

```
((${var1}=${var2}) & (${var3}=${var4} | ${var3}=${var5}))
```

3.5.1 Parenthesized expressions

Expressions enclosed in parentheses `()` have an identical value as an expression written without parentheses. Parentheses can be nested as long as they are balanced.

3.5.2 Relational operators

Operators >, >=, <, <= are meant to compare only strings that can be converted to integer values. If strings can't be converted to an integer, using these operators can produce undesirable results.

Operator	Use	Description
`>`	`${v1}>${v2}`	returns true if the integer value of v1 is greater than the integer value of v2
`>=`	`${v1}>=${v2}`	returns true if the integer value of v1 is greater or equal to the integer value of v2
`<`	`${v1}<${v2}`	returns true if the integer value of v1 is less than the integer value of v2
`<=`	`${v1}<=${v2}`	returns true if the integer value of v1 is less or equal than the integer value of v2
`=`	`v1=v2`	returns true if the string v1 is equal to string v2
`!=`	`v1!=v2`	returns true if the string v1 is NOT equal to string v2

3.5.3 Conditional operators

Operator	Use	Description
`&`	`cond1 & cond2`	returns true if cond1 and cond2 are both true
`|`	`cond1 | cond2`	returns true if either cond1 or cond2 are true
`!`	`!cond1`	returns true if condition cond1 is NOT true

3.5.4 Operator precedence

Precedence in evaluation is as follows:

operator	associativity
`* /`	left (highest)
`+ -`	left
`< <= > >=`	non-associative
`= !=`	non-associative
`&`	left
`|`	left
`=`	right (lowest)

3.6 Other tokens

Characters that separate the arguments of a keyword are "{", "}", "(", ")", and ",". The { } are generally used to denote statement blocks and "(", ")" generally encloses statement or function arguments, but variable calls are specified using the characters { } for different purposes. More details are given for each particular statement.

Mathematical expressions are specific to statement `#calc` and are explained with the function definition.

3.7 Statements

Statements are executed from beginning of file to end and from left to right. Control-flow structures such as functions are evaluated in place. The basic scripedit statement begins with a keyword, followed sometimes by a variable with parentheses ( ) and an statement block contained within brackets { }.

4 Semantics

Scoping behavior varies dependent on context as described herein. However, a general rule is that variables in nested scopes inside the current scope can't be modified from outside their nested scope, but variables from outer scopes can be accessed using the proper statements.

4.1 Variable declaration

Variables may be declared and set to a value using one of these two alternatives:

- `#def(name){body}`

  A new scope is created and the code contained in "body" is executed and an output string OUT is produced. At the end this new scope is destroyed.

  A variable with"variablename" is created in the current scope and its value is set to the string OUT. If a variable with same name already exists in the current scope, an error is generated.

- `#(variablename){body}`

  A new scope is created and the code contained in "body" is executed and an output string OUT is produced. At the end this new scope is destroyed.

If a variable with same name already exists in the current scope or an outer scope (any scope where the current scope is nested), its value will be overwritten with the string OUT. Otherwise, a variable with“variablename” is created in the current scope and its value is set to the string OUT.

Syntax notes: Whitespace including tabs and newlines are allowed between (variablename) and {body} in both cases. When the character “`#`” is not followed immediately by a “`(`” it is considered as a string constant.

4.2 Function declaration

Functions are a generalization of a variable and they are declared with a very smilar syntax. They differ in that functions take arguments whereas variables do not. Arguments are identifiers of variables or functions.

Functions may be declared and set using one of these two alternatives:

- `#def(functionname,arg1,arg2,..,argN){body}`

  A new scope is created and all the code contained in `{body}` is executed except parts that depend on the arguments. At the end this new scope is destroyed. The output string will be generated as soon as the function is called with arguments that can be resolved. Recursion is allowed so the same function can call itself during its definition.

  A function with `functionname` and arguments `arg1,arg2,...,argN` is created in the current scope and its code is set to the only argument-dependent code mentioned above. If a function with same name already exists in the current scope, an error is generated.

- `#(functionname){body}`

  A new scope is created and the code contained in “body” is executed, however, an output string can’t be created because the arguments can’t be resolved yet. Everything is executed except parts that depend on the arguments. At the end this new scope is destroyed. The output string will be generated as soon as the function is called with arguments that can be resolved. Recursion is allowed so the same function can call itself during its definition.

  If a function with same name already exists in the current scope or an outer scope (any scope where the current scope is nested), it will be overwritten with the only-argument dependent code mentioned above. Otherwise a function with“functionname” and arguments `arg1,arg2,...,argN` is created in the current scope with the same content.

Syntax notes: Whitespace including tabs and newlines are allowed between `(functionname)` and `{body}` in both cases. When the character "`#`" is not followed by a "`(`" it is considered as a string constant.

4.3 Variable call

The variable is fetched from the nearest current or outer scope. The constant string value stored in a variable is inserted to the text output using the following syntax:

```
${variablename}
```

Syntax notes: When the character "$" isn't followed by a "{" it is considered as a string constant.

4.4 Function call

In a function call arguments are passed by reference:

```
${functionname(${arg1}...${argN})}
```

The function is fetched from the nearest current or outer scope. Then a new scope is created, and the only-argument dependent code will be execute taking as the argument variables the variables that are specified in the function call. Then the recently created scope is destroyed. In order to avoid bad programming practices, variables of the outer scope which aren't specified as arguments of the function won't be accessible by the scope of the function.

Static strings are also accepted as arguments. For each static string a temporal variable is created and passed by reference to the function. However even if it may be modified, it won't be accessible by the programmer. In the following example two temporal variables are created internally by the compiler with value "hello" and "world".

```
${functionname(hello,world})}
```

4.5 File include

The following options are available:

- `#file{filename}`

  The contents of the file are inserted inline and executed without creating a new scope.

- `#file{filename,variablename}`

  or

  `#file{filename,functionname}`

  If `variablename` or `functionname` are defined in global scope of the file specified by `filename`, their definition is inserted inline and executed without creating a new scope if they don't call any variable or function out of their scope. If they call a function or variable out of their scope or they simply don't exist in the global scope of the file an error is generated.

4.6 File execution

The following options are available:

- `$file{filename}`

  A new scope is created and the content of the file is executed. At the end the newly created scope is destroyed. Again, in order to avoid bad programming practices, variables of the outer scope won't be accessible inside this scope.

- `$file{filename,variablename}`

  or

  `$file{filename,functionname(${argument1}...${argumentN})}`

  If `variablename` or `functionname` are defined in the global scope of the file, a new scope is created, their code is executed, their content goes to the output and the newly created scope is destroyed as long as they don't call any variable or function out of their scope (only arguments passed by reference to functions are accessible). If they directly call a function or variable out of their scope or they simply don't exist in the global scope of the file an error is generated.

4.7 #if and #else

`#if(condition){body}`: Evaluates *condition* if condition evaluates true, it creates a new scope executes *body* and then destroys the created scope.

```
#if(condition)
{body}
```

If `#if(condition){body}` is followed by an `#else{body2}` when condition evaluates false, another scope is created, `body2` is executed and the scope is destroyed. The `else` ambiguity produced in nested `#if` statements is resolved by connecting `#else` with the last encountered elseless `#if`.

Syntax notes: Whitespace including tabs and newlines are allowed between `(condition)` and `{body}` in both cases.

4.8 #while, #continue, #break

```
#while(condition)
{body}
```

During this loop control, a new scope is created. As long as the condition evaluates to true, the body is continually executed. The optional `#break` statement will terminate the nearest associated `#while`. Control is passed to the statement following the terminated statement. The optional `#continue` statement will jump to the the point where the nearest associated `#while` starts. Upon termination, the newly created scope is destroyed.

Syntax notes: Whitespace including tabs and newlines is optionally allowed between `(condition)` and `{body}` but ignored by scriptEdit.

4.9 #for, #next, #do

```
#for{body1}
#next{body2}
#next{body3}
...
#next{bodyN}
#do{body}
```

A new scope is created, then `body1` is executed, then `body` is executed, then `body2` is executed, then again `body`, then `body3` then `body`... For each statement block specified with `#for`{$body_i$} and then with `#next`{$body_j$}, those are executed followed by the statement block specified by `#do`{`body`} .

4.10 #exec

```
#exec(command){stdinput}
```

Execute the command exactly as given in string *command* emulating the command line. The stdinput is fed into the program as would normally be specified by the < operator. The standard output of the command is added to the output file of ScriptEdit. The standard error is ignored.

This command allows interoperability between the program and the operating system. Only a one-line command argument may be inserted here, but an arbitrary number of exec commands may be computed sequentially as needed.

Syntax notes: Whitespace including tabs and newlines are allowed between `(command)` and `{body}` in both cases.

4.11 #calc

`#calc(mathematical expression)`

Attempts to evaluate the arguments of the expression as integers, compute the result and put it back into a string. Supports basic arithmetic $(+, -, *, /)$.

Variables within the mathematical expression are converted into integer values, computed, and then returned as a string representing the integer result. If this conversion is not possible, `#calc` may return an undesired result.

Example: `#(salary){1000}#calc{salary*2}` will output a string "2000".

Syntax notes: Mathematical expressions consist of exactly two string variables or constant strings separated by one of the following binary operators $(+, -, *, /)$.

4.12 #gettoken

The following syntaxes are accepted:

`#gettoken{body}`

or

`#gettoken(DELIMCHARACTERS){body}`

Extracts from body the first token delimited by string containing all DELIMCHARACTERS. If DELIMCHARACTERS aren't specified, whitespaces, tabs, or newlines act as a delimiter. This is the way to iterate through lists (stored as ordinary string variables). This token is added to the output of ScriptEdit.

4.13 #write

`#{write(filename){body}`

A new scope is created. Opens a new file `filename` and writes the output string generated by the execution of `body` to it. The newly created scope is destroyed. If `filename` already existed, its content is overwritten.

5 Examples

Of course, some of us learn best by example.

5.1 HELLOWORLD library

```
#def(helloworld,la)
{#if(la=English){Hello world!}
#if(la=French){Bonjour monde!}
#if(la=Catalan){Bon dia mn!}
#if(la=Spanish){Hola mundo!}
#if(la=German){Guten Tag, Welt!}
#if(la=Russian){Zdravstvytye, mir!}
#if(la=Japan){Sekai e konnichiwa!}
#if(la=Latin){Orbis, te saluto!}}
```

5.2 Create an array repeating a constant value

```
#def(CreateArray,arraySize,defaultVal)
{#(count)(1)#while(${count}!=${arraySize}){${defaultVal}
 #(count){#calc(${count}+1)}}
```

5.3 Personal website in two languages

```
#for{
#(la){es}
#(color){#FFFFFF}
}
#next{
#(la){en}
#(color){#FF0000}
}
#do{
#write(index_${la}.html){
<html>
<body background="${color}">
$file{header.sehtml}
<table>
<tr>
<td><img src="icon.gif"></td>
<td>
#if(${la}=es){Bienvenidos a mi pagina personal}
#if(${la}=en){Welcome to my personal website}
</td>
</tr>
</table>

#if(${la}=es){Fotos:}
#if(${la}=en){Pictures:}
<br>

#(files){#exec(ls *.jpg)}
#while (${files}!=)
{ #(pic){#gettoken{${files}}}
  #exec(convert ${pic} -resize 400x20 -o o${pic})
  <img src="o${pic}" alt="${pic}"><br>
}

$file{footer.sehtml}
</body>
</html>
}
}
```

5.4 LaTeX article file from matlab simulations

```
#exec(matlab -e simulations(12))

% Executes simulations with parameter 12 in matlab
% Those simulations output some pictures and results
% If the user wants to change the matlab code or the
% simulation parameter
% the report will be updated automagically

% Function that inserts a picture using ScriptEdit
% No need to manually convert from eps to pdf every
% time that a new file is created...

#def(insertpicture,pic)
{#exec(epstopdf ${pic}.eps)
\includegraphics[scale=0.33]{${pic}}
}

\documentclass[a4paper,11pt]{report}
\usepackage{graphicx}
\usepackage{subfigure}
\title{A novel algorithm for Y}
\author{X, Columbia University}
```

```
\makeindex
\begin{document}

\maketitle

\section{Algorithm}

Let x,y be the input data arrays of...
% Description of the algorithm, bla, bla, bla...

\section{Results}

\begin{figure}[h!]
\begin{center}
\subfigure[data1]{${insertpicture,data1}}
\subfigure[data2]{${insertpicture,data2}}
\end{center}
\caption{Results for data1 and data2}
\end{figure}

Error for each database:

#(errors)={$file{error.txt}}
#(databases)={$file{database.txt}}
% let's suppose that matlab generates an "error.txt"
% file with the error of the algorithm in each database
% specified in file "database.txt"

\begin{center}
% use packages: array
\begin{tabular}{l|l}
DATABASE & ERROR  \\
#while(${errors}!=)
{\hline
 #gettoken{${errors}} & #gettoken{${databases}} \\
}
\hline
\end{tabular}
\end{center}

\end{document}
```