Generic Programming assignment 2

Jens Rasmussen Anders H. Pedersen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark

Abstract. In this paper we explore the syntax and uses of the macro processing tool `m4`. Our aim is to provide the reader with enough information to be able to use the tool's most principal features right away. References to more complete coverings will also be given.

Introduction

`m4` is a macro processing language, and as such, when fed input, will output a modified copy based on commands given in the input for expansion and manipulation. Its principal features are:

- String manipulation
- File insertion
- Use of system calls
- Integer arithmetic
- Conditionals

`m4` is found on most UNIX/Linux systems, and has various implementations; we will focus on GNU's version, which has also been ported to Windows.

Invoking the tool can be done as follows:

```
> m4 inputfile
```

where output is directed to standard output, the inputfile is usually named `filename.m4`.

Macro names and quoting

Macros are defined either on the command line with the `-D` option:

```
> m4 -Dmacro_name=body inputfile
```

or with a `define` command:

```
define(macro_name,body)
```

Written 3 May 2007.

which will expand all occurences of `macro_name` to `body`. Macro names must consist of alphanumeric characters and the underscore, first letter must be a nondigit. Macros are parsed by making a token out of all character sequences that are surrounded by characters not usable in a name (i.e. longest valid sequences). To avoid unwanted expansion, quoting is used:

```
define(M,B)
define(B,‘M’)
```

This evaluates `B` to `M` instead of `B`, as would be the case of quotes were omittet. `m4` removes one level of quotes under evaluation thus ‘‘`B`’’ evaluates to ‘`B`’. Output of macro evaluation is continually rescanned until no more macros are identified. (There are some pitfalls here so the reader should see [1]).

File inclusion

To include a file the following commands are used:

`include(file)` includes `file`, throws error if non-existant.

`sinclude(file)` as `include`, only errors are not reported.

Arguments to macros

Arguments are passed to macros thus:

```
define(M,‘$0 called with: $1 and $2’)
M(arg1,arg2)
```

Which outputs `M called with: arg1 and arg2`, calling the macro with fewer arguments than supported evaluates the missing `$n` to the empty string and calling it with more just ignores the unmatched arguments.

Conditional expressions

We describe two of the five built-in conditionals (rest deal with iteration and recursion, see [1]):

`ifdef(macro,do_if_defined[,else])` where `else` is optional.

`ifelse(s1,s2,if_equal[,else])` does if_equal if strings `s1` and `s2` are equal, the `if_equal` part can have the form `if_equal,s’1,s’2,if_equal’` if the latter is used the optional `else` must be omitted.

Example:

```
define(s1,‘a’)
define(s2,‘b’)
ifelse(s1,s2,‘if_equal’,s2,s2,‘else’)
```

In this example the first string comparison returns not equal, so the remaining three arguments are processed in the same way and yield `else`.

Integer arithmetics

Integer arithmetic is built in through the command,

```
eval(expression,[radix],[width])
```

where `radix` and `width` are optional. The argument `expression` is a mathematical expression using C-like syntax, that may consist of any of the basic mathematical operators, logical operators, bitwise operations and shifts. A complete list of operators can be found in [1]. A number in `expression` can be represented as a decimal, octal, hexadecimal, binary and radix. Exactly how these are represented can be seen in [1].

The optional `radix` argument in the `eval` expression, specifices the radix to be used in the expansion, where 10 is default, and it must be between 0 and 36. The `width` argument describes the minimum output width.

Example:

```
eval(‘2 ** (0r2:0111 + 0b1000 - 015)’, 2, 4)
```

In this example, 2 is raised to the exponent in the paranthesis. This exponent is calculated using three different number representations, i.e. radix (base 2), binary and octal. The output of the example is `0100`, radix (also base 2), with its minimum width of 4 characters.

String manipulation

String manipulation is handled by the built-in commands (all indexing starts from zero):

`len(s)` length of `s`.

`index(s,substr)` index of `substr` in `s`, returns -1 if unmatched.

`regexp(s,regexp[,replacement])` searches for and optionally replaces with `replacement` first match of regular expression `regexp` in `s`, returns index or replacement of match, returns -1 if unmatched.

`substr(s,start_index[,length])` extracts substring of `s` from `start_index` and `length`-1 characters forward, or to end of `s` if `length` is omitted.

`translit(s,chars[,replacement])` deletes occurrences of characters `chars` or replaces them with other characters, where each character index in `chars` is matched to corresponding index in `replacement`.

`patsubst(s,regexp[,replacement])` deletes or replaces with `replacement` all matches of regular expression `regexp` in `s`, returns the modified string.

`format(s,...)` basically a form of C's `printf`, see [1].

The regular expressions use GNU Emacs's syntax, see [2].

Example:

```
regexp(‘This is it!’,‘\<[i]’)
patsubst(‘This is it!’,‘\<[i]’)
```

`regexp` returns `5`, while `patsubst` returns `This s t!`

Example:

```
translit(‘tmf’,‘a-z’,‘Z-A’)
```

Which returns `GNU`.

System commands

System shell-commands can utilized with the following commands:

`syscmd(cmd)` calls the shell command `cmd`, it does not process (and thus expand) its output.

`esyscmd(cmd)` as `syscmd`, but output is processed.

`sysval` returns the exit status of the last `syscmd` or `esyscmd` run, returns 0 of none have been run.

`maketemp(s)` creates a temporary file named `s`, is should be postfixed by six `X`'s which the command replaces by six randomly generated characters to create a unique id. (If the `X`'s are omitted, the six random will be concatenated to `s`.).

Example:

```
define(‘echo’,‘‘echo’ ‘echo’’)
sys: syscmd(echo) | esys: esyscmd(echo)
```

will result in: `sys:  echo | esys:  echo echo` as the output of the latter is processed but not that of the first.

References

[1] The GNU Project, Gnu m4 manual, Worldwide Web Document (2007). Available at `http://www.gnu.org/software/m4/manual/`.

[2] The GNU Project, Gnu emacs manual, Worldwide Web Document (2007). Available at `http://www.gnu.org/software/emacs/manual/`.