

DollarMacro: General Purpose Macro Expander

Andrew Rock
School of Information and Communication Technology
Griffith University
Nathan, Queensland, 4111, Australia
a.rock@griffith.edu.au

March 25, 2009

Abstract

This document lists and describes a Haskell implementation of a simple macro expander. ¹

1 Introduction

Composing HTML pages can be made easier if commonly repeated text doesn't have to be commonly repeated. By defining these sequences as reusable, parameterized macros the pages become quicker to create and easier to maintain.

This document lists and describes a Haskell implementation of a simple macro expander. The design is such that other languages other than HTML could also be preprocessed with this tool.

2 Obtaining and building

DollarMacro is available for download as a haskell sources from <http://www.cit.gu.edu.au/~arock/DollarMacro/DollarMacro.tar.gz>. To build it, you need a Haskell compiler (GHC is recommended) from <http://www.haskell.org>.

To prepare for compilation on a UNIX-like system, change directory to `DollarMacro/src`.

```
$ cd DollarMacro/src
```

To compile the DollarMacro tool, type:

```
$ make bin
```

3 Users Guide

3.1 Input file format

The input file can have any name and extension. It may contain any arbitrary text, marked up with macro definitions and calls.

¹DollarMacro – a language independent macro expander Copyright (C) 2005, Andrew Rock

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

3.2 Macro markup symbols

All special character sequences begin with a dollar sign (\$), hence the name of the tool. The sequences are:

```
$=  introduce a new macro;
$(  begin a macro parameter list;
$,  separate macro parameters;
$)  end a macro parameter list;
${  begin a macro definition;
$}  end a macro definition;
$+  call a macro or parameter;
$-  undefine a macro or parameter;
$<  introduce the name of a file to include;
$>  end the name of an included file;
$?  introduce text to expand only if a macro is defined;
$!  introduce text to expand only if a macro is not defined;
$[  begin a comment;
$]  end a comment;
$#  rub out the following character; and
$$  output a single $.
```

Macro and parameter names are sequences of one or more of letters and digits. Examples: `a`; `A`; `andrew`; `Andrew`; `ANDREW`; `42`; and `andrew42`. These names can be distinguished from ordinary text by their context.

Whitespace is very significant. None should appear between markup symbols.

3.3 Macro markup examples

This example contains a comment and some spurious dollar signs. A dollar sign that is not followed by another significant character is output as normal. A sequence of two dollar signs is output as one dollar sign, allowing special macro sequences to be output as plain text.

```
I have $3.20 in my pocket.
I have $[only$] $$3.20 in my pocket.
Note that $$[ starts a comment, terminated with $].
```

Output:

```
I have $3.20 in my pocket.
I have $3.20 in my pocket.
Note that $[ starts a comment, terminated with $].
```

This example contains an include directive.

```
A short$<Test2.inc$>file.
```

This is the file to include.

```
, but quite instructive,
```

Output:

```
A short, but quite instructive, file.
```

This example defines a macro and calls it. Note the use of the rub out directive to suppress a spurious line ending.

```
$=name${Andrew$}$#
My name is $+name.
```

Output:

```
My name is Andrew.
```

This example shows the use of a parameter.

```
$=name${Andrew}$##
$=sentence$(who$){My name is $+who.$}$#
$+sentence$($+name$)
```

Output:

```
My name is Andrew.
```

This example demonstrates the conditional directives.

```
$=flag${}$##
flag $?flag${was defined}$!flag${was not defined}.
```

Output:

```
flag was defined.
```

This example demonstrates the use of file inclusion with a macro call in the file name.

```
$=testNumber${6}$##
A slightly longer$<Test$+testNumber.inc$>file.
```

This is the file to include.

```
, but even more instructive,
```

Output:

```
A slightly longer, but even more instructive, file.
```

Macro calls in include file names may not have arguments. `##` may be used to delimit the end of a macro name in include file names.²

3.4 Built-In Macros

Some operations are easier to implement in Haskell than as macros, so these operations are provided as built-ins. They are documented as follows:

`##DateAndTime` inserts the current date and time.

`##EncodeHTMLSpecials$(text$)` encodes the special characters that delimit html tags in *text* as entities so that any arbitrary text may be included in a HTML document.

`##IfNotBlank$(testText$,text$)` outputs *text* iff *testText* contains any non-whitespace characters.

`##SelectColumns$(columns$,table$)` selects the specified *columns* from the *table*. The *columns* are specified as a list of integers in square brackets; e.g. `[0,2]`. Columns are numbered from 0. Columns in *table* are delimited by whitespace.

`##Trim$(text$)` strips the leading and trailing whitespace from *text*.

These built-in macro names are predefined but not reserved. If you define macros with the same name they will eclipse the built-in.

3.5 Using the tool

Use the tool from the command line.

DollarMacro *file*

A single command line argument specifies the source *file* name. If the command line argument is omitted, or it is “-” then standard input is read. Output always goes to standard output.

If using Hugs, use the expression `run path` to process the file at *path*.

²Use this trick wherever a macro call with no arguments must run to other text that might be interpreted as part of the macro name.

4 Implementation

4.1 Preliminaries

module Main (main) where

```
import Prelude hiding (error); import Time; import System; import IO
import Char
```

```
import ABR.CGI; import ABR.BSTree; import ABR.List; import ABR.String
```

4.2 Processing the source text

This is the stand-alone entry point.

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    []      -> run "-"
    path : _ -> run path
```

This is the hugs entry point. `run path` will process the source at `path`.

```
run :: FilePath -> IO ()
run path = do
  source <- case path of
    "-" -> getContents
    _   -> readFile path
  (output,_) <- expand source emptyBST
  putStr output
```

4.3 A macro definition

A macro or parameter name and its definition are just strings. A macro, apart from its name consists of the names of its parameters and its definition. A symbol table is a mapping from names to macros.

```
type Name = String
type Definition = String
type Macro = ([Name], Definition)
type Table = BSTree Name Macro
```

`addMacro name macro table` adds the *named macro* to the *table* of macro definitions.

```
addMacro :: Name -> Macro -> Table -> Table
addMacro = updateBST (\x _ -> x)
```

4.4 Macro expansion

`expand source table` returns the expansion of the *source* using the *table* of macro definitions.

```
expand :: String -> Table -> IO (String, Table)
expand source table = case source of
  "" -> return ("", table)
  c:cs -> case c of
    '$' -> dollar cs table
    _   -> do
      (cs',table') <- expand cs table
      return (c : cs', table')
```

`dollar source table` returns the expansion of the *source* using the *table* of macro definitions, given that the previous character was a dollar sign.

```

dollar :: String -> Table -> IO (String, Table)
dollar source table = case source of
  "" -> return ("$", table)
  c:cs -> case c of
    '=' -> define cs table []
    '+' -> lookupMacro cs table []
    '-' -> undefine cs table []
    '<' -> include cs table []
    '?' -> ifDef cs table []
    '! ' -> ifNotDef cs table []
    '[' -> comment cs table
    '#' -> expand (if null cs then "" else tail cs) table
    '$' -> do
      (cs',table') <- expand cs table
      return ('$' : cs', table')
    _ -> do
      (cs', table') <- expand cs table
      return ('$' : c : cs', table')

```

`define source table name` returns the expansion of the *source* using the *table* of macro definitions, given that the previous characters indicate that we are defining a new macro and the name is being read. *name* is the accumulated name so far, in reverse.

```

define :: String -> Table -> Name -> IO (String, Table)
define source table name = case source of
  [] ->
    error' "Source ends inside macro name"
  [c] ->
    error "Source ends inside macro name" source
  '$' : '(' : cs ->
    defArgs cs table (reverse name) [[]]
  '$' : '{' : cs ->
    defBody cs table (reverse name) []
  c : cs ->
    if isAlphaNum c then
      define cs table (c : name)
    else
      expand source (addMacro name ([],"") table)

```

`defArgs source table name argNames` returns the expansion of the *source* using the *table* of macro definitions, given that the previous characters indicate that we are defining a new macro and the name is *name*. *argNames* is the accumulated list of argument names so far in reverse.

```

defArgs :: String -> Table -> Name -> [Name] -> IO (String, Table)
defArgs source table name args = case source of
  [] ->
    error' "Source ends inside arg list"
  [c] ->
    error' "Source ends inside arg list"
  '$' : ',' : cs ->
    if not (null (head args)) then
      defArgs cs table name
        ([] : reverse (head args) : tail args)
    else
      error source "Missing arg name"
  '$' : ')' : '$' : '{' : cs ->
    if not (null (head args)) then
      defBody cs table name (reverse (head args) : tail args)
    else
      error source "Missing arg name"
  '$' : ')' : cs ->

```

```

    if not (null (head args)) then
      expand cs (addMacro name ((reverse (head args))
        : tail args), "") table)
    else
      error source "Missing arg name"
  c : cs ->
    if isAlphaNum c then
      defArgs cs table name ((c : head args) : tail args)
    else
      error source "Unexpected text in arg name"

```

`defBody source table name argNames` returns the expansion of the *source* using the *table* of macro definitions, given that the previous characters indicate that we are defining a new macro and the name is *name*. *argNames* is the accumulated list of argument names. We are about to collect the body of the macro.

```

defBody :: String -> Table -> Name -> [Name] -> IO (String, Table)
defBody source table name args = do
  (body, cs) <- collectGroup source ""
  expand cs (addMacro name (args, reverse body) table)

```

`collectGroup source out` returns the text delimited by a closing `$}` and the remaining text. The group may contain nested `#{ $}` pairs. *out* is the group collected so far.

```

collectGroup :: String -> String -> IO (String, String)
collectGroup source out = case source of
  []          -> do
    error' "Source ends inside group"
    return undefined
  [_]        -> do
    error' "Source ends inside group"
    return undefined
  '$' : '{' : cs -> do
    (out',cs') <- collectGroup cs ""
    collectGroup cs' ("}$" ++ out' ++ "{$" ++ out)
  '$' : '}' : cs -> return (out, cs)
  c : cs      -> collectGroup cs (c : out)

```

```

lookupMacro :: String -> Table -> Name -> IO (String, Table)
lookupMacro source table name = case source of
  []          ->
    expandMacro source table (reverse name) []
  '$' : '(' : cs ->
    lookupArgs cs table (reverse name) [[]]
  c : cs ->
    if isAlphaNum c then
      lookupMacro cs table (c : name)
    else
      expandMacro source table (reverse name) []

```

```

lookupArgs :: String -> Table -> Name -> [String] -> IO (String, Table)
lookupArgs source table name args = case source of
  []          ->
    error' "Source ends inside arg list"
  '$' : ',' : cs -> do
    (arg,table') <- expand (reverse (head args)) table
    lookupArgs cs table' name ([] : arg : tail args)
  '$' : '(' : cs -> do
    (out,cs') <- collectArg cs "$"
    lookupArgs cs' table name ((out ++ head args) : tail args)
  '$' : ')' : cs -> do

```

```

    (arg, table') <- expand (reverse (head args)) table
    expandMacro cs table' name (arg : tail args)
c : cs      ->
    lookupArgs cs table name ((c : head args) : tail args)

collectArg :: String -> String -> IO (String, String)
collectArg source out = case source of
  []      -> do
    error' "Source ends inside nested arg"
    return undefined
  [_]    -> do
    error' "Source ends inside nested arg"
    return undefined
  '$' : '(' : cs -> do
    (out',cs') <- collectArg cs "$"
    collectArg cs' (out' ++ out)
  '$' : ')' : cs -> return (")$" ++ out, cs)
c : cs      -> collectArg cs (c : out)

expandMacro :: String -> Table -> Name -> [String] -> IO (String, Table)
expandMacro source table name args = do
  let mMacro = lookupBST name table
      case mMacro of
        Nothing ->
          if isBuiltIn name then
            expandBuiltIn source table name (reverse args)
          else
            error source ("Undefined macro: " ++ name)
        Just (as, b) ->
          if length args /= length as then
            error source ("Number of arguments for macro " ++
              name ++ " do not match")
          else do
            let cs = expand' b (zip as args)
                (cs',table') <- expand cs table
                (source',table'') <- expand source table'
            return (cs' ++ source', table'')

expand' :: String -> [(Name,String)] -> String
expand' body args = case body of
  []      -> []
  '$' : '+' : cs ->
    let name = takeWhile isAlphaNum cs
        cs' = dropWhile isAlphaNum cs
        mVal = lookup name args
    in case mVal of
      Nothing -> "$+" ++ name ++ expand' cs' args
      Just val -> val ++ "$# " ++ expand' cs' args
c : cs      -> c : expand' cs args

undefine :: String -> Table -> Name -> IO (String, Table)
undefine source table name = case source of
  []      ->
    error' "Source ends inside macro name"
c : cs ->
  if isAlphaNum c then
    undefine cs table (c : name)
  else
    expand source (deleteBST name table)

include :: String -> Table -> String -> IO (String, Table)
include source table path = case source of

```

```

[]          ->
  error' "Source ends inside include"
[c]         ->
  error' "Source ends inside include"
'$' : '#' : c : cs ->
  include cs table path
'$' : '+' : cs -> do
  let (name,cs') = span isAlphaNum cs
      mMacro = lookupBST name table
  case mMacro of
    Nothing -> error source
              ("Undefined macro in include file name: " ++ name)
    Just (as, b) -> if length as /= 0 then
                    error source
                    "Macros in include file names may not have \
                    \args."
    else
      include (b ++ cs') table path
'$' : '>' : cs -> do
  source' <- readFile (reverse path)
  (source'', table') <- expand source' table
  (cs',table'') <- expand cs table'
  return (source'' ++ cs', table'')
c : cs ->
  include cs table (c : path)

ifDef :: String -> Table -> Name -> IO (String, Table)
ifDef source table name = case source of
[]          ->
  error' "Source ends inside ifDef macro name"
'$' : '{' : cs -> do
  (block, cs') <- collectGroup cs []
  if memberBST (reverse name) table then do
    (block', table') <- expand (reverse block) table
    (cs'', table'') <- expand cs' table'
    return (block' ++ cs'', table'')
  else
    expand cs' table
c : cs ->
  if isAlphaNum c then
    ifDef cs table (c : name)
  else
    error source "Missing ifDef block"

ifNotDef :: String -> Table -> Name -> IO (String, Table)
ifNotDef source table name = case source of
[]          ->
  error' "Source ends inside ifDef macro name"
'$' : '{' : cs -> do
  (block, cs') <- collectGroup cs []
  if not (memberBST (reverse name) table) then do
    (block', table') <- expand (reverse block) table
    (cs'', table'') <- expand cs' table'
    return (block' ++ cs'', table'')
  else
    expand cs' table
c : cs ->
  if isAlphaNum c then
    ifNotDef cs table (c : name)
  else
    error source "Missing ifDef block"

```

```

comment :: String -> Table -> IO (String, Table)
comment source table = case source of
  []      ->
    error' "Source ends inside comment"
  [c]    ->
    error' "Source ends inside comment"
  '$' : ']' : cs ->
    expand cs table
  _ : cs   ->
    comment cs table

```

4.5 Built-in special macros

`isBuiltIn name` returns `True` iff `name` is one of the names of the predefined macros implemented in Haskell.

```

isBuiltIn :: Name -> Bool
isBuiltIn name = name `elem` [
  "TimeAndDate",
  "EncodeHTMLSpecials",
  "SelectColumns",
  "Trim",
  "IfNotBlank"
]

```

`expandBuiltIn source table name args` expands the built-in macro `name` using `args` then continues to expand `source` using `table`.

```

expandBuiltIn :: String -> Table -> Name -> [String] -> IO (String, Table)
expandBuiltIn source table name args = do
  let e cs = do
        (source',table') <- expand source table
        return (cs ++ source', table')
      msg0 = "Built-in macro not implemented"
      msg1 m = "Wrong number of arguments for built-in macro " ++ m
  case name of
    "EncodeHTMLSpecials" -> case args of
      [cs] -> e (makeHTMLSafe cs)
      _     -> error source (msg1 "EncodeHTMLSpecials")
    "TimeAndDate" -> case args of
      [] -> do
        ct <- getClockTime
        ct' <- toCalendarTime ct
        e (calendarTimeToString ct')
      _ -> error source (msg1 "TimeAndDate")
    "SelectColumns" -> case args of
      [cols,table] -> e (selectColumns cols table)
      _             -> error source (msg1 "SelectColumns")
    "Trim" -> case args of
      [cs] -> e $ trim cs
      _     -> error source (msg1 "Trim")
    "IfNotBlank" -> case args of
      [cs,ds] | all isSpace cs -> e ""
              | otherwise      -> e ds
      _             -> error source (msg1 "IfNotBlank")
    _ ->
      error source msg0

```

```

selectColumns :: String -> String -> String
selectColumns cols table =
  let is :: [Int]

```

```
is = read $ trim cols
xss = map words $
      filter (not . null) $
      map trim $
      lines table
in unlines [unwords [xs !! i | i <- is] | xs <- xss]
```

4.6 Messages

```
error :: String -> String -> IO (String, Table)
```

```
error source msg = do
  hPutStrLn stderr $
    "\n\nError: " ++ msg
  ++ ".\n\nAt:\n\n" ++ take 200 source
  return ("", emptyBST)
```

```
error' :: String -> IO (String, Table)
```

```
error' msg = do
  hPutStrLn stderr $
    "\n\nError: " ++ msg ++ "."
  return ("", emptyBST)
```