rPrepro

—

A Simple Text Preprocessor Handling Conditional Text and Macros

Dr. Jürgen Vollmer

November 27, 2008
Version 2008/11/27

Abstract

rPrepro is a simple preprocessor handling conditional text and text macros.

This preprocessor uses uppercase letters for the preprocessor directives, so that in can be called before another preprocessor e.g. *CPP*(1) without things are intermixed. For example, a PUMA specification can be processed with *rPrepro* before the result is given to PUMA, which produces C-code, which is then processed by *CPP*(1).

1 Synopsis

rPrepro [*-chlnpsvV*] [**-d***name*] [**-D***name*] [**-D***name*=*value*] [**-P***name*] [**-P***name*=*value*] [**-U***name*] [**-I***path*] [**-o***outfile*] [**-t**[*text*]] [*infile*]...

2 Description

rPrepro evaluates the preprocessor directives and conditional grammar rules as defined below. *rPrepro* reads *infile*s or `stdin` and writes the ouput to *outfile* or `stdout`.

2.1 Directives

A preprocessor directive starts at the beginning of a line with a # which is followed by the directive name. Before and after the # blanks are allowed. A directive is terminated by a newline. The following directives are processed:

- `#IF` *expression*
- `#IFDEF` *identifier*
- `#IFNDEF` *identifier*
- `#ELIF` *expression*
- `#ELIFDEF` *identifier*
- `#ELIFNDEF` *identifier*

- `#ELSE`
- `#ENDIF`
- `#//` *the rest of the line is a comment*
- `#ERROR` *write the rest of the line to stderr and set exit code 2, if the directive is contained, else ignore the directive in source which is not skipped*
- `#MESSAGE` *write the rest of the line to stderr, if the directive is contained in source which is not skipped, else ignore the directive*

Where *identifier* must start with a letter and followed by a sequence of letters, digits, underscores (_), hyphens (-) or dots (.). *identifier*s are case sensitive. The last character of an identifier must be either a digit or a letter. *version* is a (version) number constisting of an optional + or − followed by a sequence of digits and dots. *filename* is any character sequence without a `"` and newline. A *string* is any character sequence enlcosed in `"`, including newline but without unescaped `"`. If `"` should be part of the string, it must be escaped as `\"`.

The directives must be propperly nested and have the usual semantics (see e.g. the C preprocessor *CPP*(1)), by selecting the text which should be emitted or skipped.

Expressions have the form:

- `DEFINED (` *identifier* `)`
- *identifier*
- *identifier op version*
- *identifier op string*
- *expression* `&&` *expression*
- *expression* || *expression*
- `^` *expression*
- `(` *expression* `)`

Expressions are evaluated to a boolean value as following:

- The precedence of the boolean operators `^` (boolean negation), `&&` (boolean and) and || (boolean or) is: `^` is higher than `&&` is higher than ||.
- The accepted relational (string compare) operators *op* are: `== = != < <= > >=`.
  = is a short hand for ==.
- *version* numbers are treated as string constants.
- The value of an *identifier* used as operand of a relational expression is the value set by a command line option (**-D**, **-d** or **-P**) or by a `DEFINE` directive.
- All all comparisions between *identifier*s and *string*s or *version* numbers are done using the C-library function `strcmp()`.
- The expression consisting only of an *identifier* is a shorthand for `DEFINED (` *identifier* `)`.
  Evaluating this expression results *true* iff the *identifier* is defined (by a commandline option or `DEFINE` directive).

2.2 Trailing backslash

If a line containing one of the directives `IF, IFDEF, IFNDEF, ELIF, ELIFDEF, ELIFNDEF, ELSE` or `ENDIF` is terminated by a backslash (\), that line as well as the skipped lines are terminated by a trailing \(if the source lines are termined by a \). This ensures correct processing of e.g. CPP.
For example assume the following source:

```
# define a(x)      \
# IF foo           \
            b (x)  \
# ELSE             \
            c (x)  \
# ENDIF            \
            d (x)
```

The result after processing (if `foo` is not a defined macro) is:

```
# define a(x)      \
\
\
\
            c (x)  \
\
            d (x)
```

Without terminating the directives by a \, the result would look like:

```
# define a(x)      \



            c (x)  \

            d (x)
```

Which will cause e.g. CPP do the wrong processing.

2.3 Defining Names

The identifiers used in the expressions of conditional grammar rules as well as of directives may be defined in several ways:

- The command line options **-d***name*, **-D***name* and **-D***name*=*value*
- The command line options **-P***name* and **-P***name*=*value*
- The command line option **-p**. This option does not define any name, but it lets all expressions of the condtional grammar rules and the conditional `!INCLUDE`s always to evaluate to true.
- The `#DEFINE` directive:
  - `#DEFINE` *identifier*

  - `#DEFINE` *identifier* = *value*

- The `!DEFINE` directive:
  - `!DEFINE` *identifier*
  - `!DEFINE` *identifier* = *value*

- A definition may be canceled by the `#UNDEF` / `!UNDEF` directive (both forms are equivalent):
  - `#UNDEF` *identifier*
  - `!UNDEF` *identifier*

Where *value* is either a string or version constant as descibed above.

The difference between `#DEFINE` and `!DEFINE` is that `#DEFINE`ed names will be replaced in the source by their given value, while `!DEFINE`ed names will not be replaced in the source:

If the source contains a `#DEFINE`d name, that identifer is replaced by the value given in the `#DEFINE` directive. That replacement takes place almost everywhere in the source: except in strings and comments. in the source. The macro expansion is not recursive, i.e. if the *value* contains an identifier defined elsewhere, that identifier is not expanded in *value*. If a defined name gets no value assigned, it has as implicitly assigned value the empty string.

Definitions and undefinitions are valid from the point in the source, where they occur and stay valid until they are redefined.

Names defined using the **-d** or **-D** option behave like `#DEFINE`d names.

Names defined using the **-P** option behave like `!DEFINE`d names.

Definitions (**-D/P***name*, **-D/P***name*=*value*) or undefinitions (**-U***name*) of names given on the command line override any `DEFINE` or `UNDEF` directives contained in the source for the same identifier.

If several files are given on the command line, they run in the same ”environment”, i.e. the `DEFINE`s etc. from a first file are valid in a second file.

2.4 Conditional Grammar Rules

rPrepro knows a second kind of conditional text selection, the so called *Conditional Grammar Rules*. They are used e.g. in parser specifications. The conditional grammar rule starts with a `!` at the beginning of a line and ends with the dot terminating the parser grammar rule. The selection condition is enclosed in a pair of `!` (Note: leave a blank after the closing `!` to distinguish it from e.g. `!=` used as condition).

Conditional grammar rules are processed only if at least one name is defined using the **-P** command line option, or if **-p** is given. If none is given, conditional grammar rules are not processed at all, with other words: `!`*condition*`!` clauses are passed unchanged to the output!

```
!DB2! = CREATE FOO name .
!OC!  = CREATE FOO name WITH bar .
```

If the command line option [**-P***DB2*] is given in this example, then the text following the second `!` is emitted up to and including the trailing dot, while the line starting with `!OC!` will be skipped.

The general syntax is:

- ! *condition* ! *grammar-rule* .
- ! *condition* ! *grammar-rule* . // *comment*
- ! *condition* ! *grammar-rule* . /* *comment* */

where *condition* is a list of *expression*s of the form:

- *condition* ; *expression*
- *condition* , *expression*
- *condition* *expression*
- *expression*

The , (comma) operator corresponds to a logical *or* of the *condition* and the *expression*. The ; (semicolon) operator corresponds to a logical *and* of the *condition* and the *expression*. The concatenation of *condition* and an *expression* is a short-hand for the , (comma) operator.

The difference between , (;) and || (&&) is the reversed operator precedence:

- && (*and*) has a higher precedence than || (*or*), which is the usual, from mathematics known precedence; while
- ; (*and*) has a lower precedence than , (*or*).
- ; and , have a higher precedence && and ||.

Therfore:

- `a ; b , c <=> a AND (b OR c)`
- `a && b || c <=> (a AND b) OR c`

This makes it easier to write conditions like:

- `DML ; DB2 OC`
  (translate this as e.g.: *"A SQL data manipulation language statement defined in the SQL-dialects DB2 and OC"*, which is equivalent to the longer variant:
- `DML && (DB2 || OC)`

grammar-rule is any legal *lpp*(1) grammar rule. If the grammar rule is a nested one (i.e. has the form < ... > the closing > and the following dot (.) must be on the same line.

The *comment* must be written on the same line as the dot.

If the *condition* evaluates to true, the text of the *grammar-rule* as well the *comment* are emitted. If the condition evaluates to false, the text of the *grammar-rule* as well the *comment* are skipped. The grammar rule (inside and outside of rule target code) may contain any other directive as described above.

2.5 File Inclusion

Files may be included using the following directives:

- #INCLUDE "*filename*"
- ! *condition* ! #INCLUDE "*filename*" .

- ! *condition* ! #INCLUDE "*filename*" . // *comment*
- ! *condition* ! #INCLUDE "*filename*" . /* *comment* */

For conditional includes (those with a leading ! *condition* !) a trailing dot is needed, in order to terminate the scope of the condition. The word #INCLUDE, the filename and the trailing dot (if given) must be on the same line. The trailing dot and the comment are not emitted, independent of the evaluated value of the condition.

rPrepro processes #INCLUDEs, by processing and copying the included file to the output. In order to keep source-file position information correct, at the beginning and at the end of each included file a line containing

```
#@ line line-nr "filename"
```

or

```
# line line-nr "filename"
```

may be created (see also **-n**).
#INCLUDEs may be nested.

Directives should not span over source files, i.e. the #IF and its #ENDIF must be part of the same file.

2.6 Notes on Strings, Comments, Target Code

rPrepro ignores the content of string literals (enlosed in double as well as single quotes), and comments (multiline C-style as well as onle-line C++-style comments) when performing handling the directives and conditional grammar rules.

Therefore if the source contains a single or double quote character which does not start / end a string literal this quoute character should be escaped by prepending a \(backslash) character.

A grammar rule usually contains some target code enclosed with braces ({ and }). Target code may not contain a conditional grammar rule (since e.g. the language C contains as token the !-character). Therfore *rPrepro* knows the concept of matching braces. As a consequence if a unpaired brace should not denote the opening or closing of target code (outside of target code), the brace must be escaped as described above. If braces occur paired, no escaping is needed.

The character sequence /* starts usually a comment, which must be closed by */. If /* should not start a comment, one has to escape it as \/* as described above.

The !, # and / may be escaped outside of strings, target code and comments.

All escaping backslash is removed by *rPrepro*.

Note: take care when using *rPrepro* to process scanner specifications: To get a scanner regaular expression \" one has to write \\\". Reason: the first two \ will be unescaped to one \ in the output, and in order to avoid that " starts a string in view of *rPrepro*, one has to esacpe it as \". Therefore: three \!

3 Options

-h Show a help text.

-l Emit line numbers.

-n Do not generate #-line directives.
Default: when a a new file is read create a #@ line directive with the following format

```
#@ line line-nr "filename"
```

-c Emit the line-directive in CPP-style, i.e. generate

```
# line line-nr "filename"
```

-s Do not emit blanks and new lines for skipped text.

-V Verbose (show filename of parsed file).

-v Show version information.

-d*name* Define the macro *name* to have the value *name* (same as **-D***name*=*name*).

-D*name* Define the macro *name* (with the empty string as value).

-D*name*=*value* Define the macro *name* to have the value *value*.

-p All conditional grammar rules are selected Note: **-p** does not imply **-D***name* for any *name*!

That's due to the fact, that *rPrepro* does not know the macro names in advance.

In order to use macro names which may be defined sometimes using **-D***name* in conditional directives like `#IF` one may add at the beginning of a source:

```
# IF ^(DB2 || OC)
  !DEFINE DB2
  !DEFINE OC
# ENDIF

# IF DB2
....
# ENDIF

!OC!  = CREATE FOO name WITH bar .
```

(given we have only those two alternatives).

-P*name* Enable conditional grammar rule selection for *name* and assign the empty string to *name*.

-P*name*=*value* Enable conditional grammar rule selection for *name* and assign *value* to *name*.

-U*name* Undefine the macro *name*.

-I*path* Search included files in *path*. Several path may be given.

-o*outfile* Write output to file *oufile*, default `stdout`.

-t[*text*] If no *text* is given, emit a default text containing the current date and time which is enclosed within `/* .. */` as first and last line of the generated output. If *text* is given, emit that *text*.

infile Read from *infile*, default: `stdin`. If several files are given, they run in the same "environment", i.e. `#DEFINE` etc. from a first file are valid in a second file too.

4 Example

Calling `rPrepro -D"A=with a" -P"DB2=7.0" -PDML -t prog.txt` for the following input file `prog.txt`

```
# IFDEF A
   this is the case where "A" is defined with value: \"A\'
# ELSE
   this is the case where "A" is not defined
# ENDIF

# IF A == "with a"
  only if "A" is defined and is equal to 'with a'
# ENDIF

!DB2,OC! foo = <
 !DB2 >  7.0! = f:foo_bar w:with_option
                {tree := mfoo_bar (f:tree,w:tree)}
              . // only newer DB2
 !DB2 <= 7.0! = f:foo_bar
                {tree := mfoo_bar (f:tree,NoTree)}
              .
              = f:foo_bar w:other_option
                {tree := mfoo_bar (f:tree,w:tree)}
              . // DB2 and OC
> .  // only in DB2 or OC

!DML; DB2, OC!
bar1 = <
   = DML for DB2 and OC, the short way
   .
> .

!DML && DB2 || OC!
bar1 = <
   = DML for DB2 and OC
   .
> .

!DML && (DB2 || OC)!
bar1 = <
   = DML for DB2 and OC, the long way
   .
> .

!DML; MS!
bar1 = <
   = DML for MS
   .
> .
```

```
!DDL; DB2, OC!
bar2 = <
   = DDL for DB2 and OC
   .
> .

!DDL; MS!
bar2 = <
   = DDL for MS
   .
> .
```

---

the result looks like:

```
/* **** THIS FILE IS GENERATED DO NOT EDIT ****
 * **** generated on Sat Apr 18 14:22:24 2009
 */
#@ line 1 "prog.txt"

   this is the case where "A" is defined with value: "with a'




  only if "A" is defined and is equal to 'with a'


         foo = <



              = f:foo_bar
                {tree := mfoo_bar (f:tree,NoTree)}
              .
              = f:foo_bar w:other_option
                {tree := mfoo_bar (f:tree,w:tree)}
              . // DB2 and OC
> .  // only in DB2 or OC


bar1 = <
   = DML for DB2 and OC, the short way
   .
> .


bar1 = <
```

```
   = DML for DB2 and OC
   .
> .


bar1 = <
   = DML for DB2 and OC, the long way
   .
> .
```

```
/* **** THIS FILE IS GENERATED DO NOT EDIT ****
 * **** generated on Sat Apr 18 14:22:24 2009
 */
```

---

5 Exit Codes

Exits codes are:

0 no error occured

1 the source contains syntax or semantic errors

2 the source contained a `ERROR` directive

3 some fatal internal error occured

6 See Also

cpp(1) The C-macro preprocessor

7 Version

Version: 2008/11/27 of November 27, 2008.

8 Changes

2007/04/27	Added `!UNDEF`, fixed a grammar problem.
2007/04/18	Fixes macro definition inside `#IF..#ENDIF`.
	Handling of trailing backslashes.
2007/01/22	Added `;` operator in conditions.
2006/12/14	Added conditional grammar rules.
2003/11/30	Added **-d***name*.
2001/04/11	Added all relational operators.
	Added `MESSAGE` and `ERROR`.
	Fixed docu.
	Compiles and runs under Microsoft
2001/04/10	Initial version.

9 Copyright

10 Author

Dr. Jürgen Vollmer
CoCoLab - Datenverarbeitung
Höhenweg 6
D-77855 Achern
Email: `vollmer@cocolab.com`
WWW: `http://www.cocolab.com`.

Contents