Parsing and how it affects formatting 9

A Formatter for C and C++ source

cdent is an MPW tool which formats C and C++ source code. By default, the format style is that of The C Programming Language by Kernighan and Ritchie. An alternative style, developed for MacApp by Michael Burbidge, is also available. Additionally, the user can specify

Places, Contexts, Commands, and Glue

Formatting commands are applied to specific *places* in the input. The places where formatting is applied are defined by cdent and name syntactic entities like opening curly braces after the condition of an “if” or the “case” label in a “switch” statement. A *context* is the range over which certain formatting commands have an effect. A context is best thought of as that which exists between opening and closing braces and parentheses. Formatting *commands* affecting the indentation have a lifetime that begins when first seen until the end of the context. *Glue* (a term stolen from Knuth’s TEX) is the formatting command sequence applied to a particular place. Glue is predefined by cdent but can be specified on the command line.

Numeric registers

®0	The global indent delta
®1	The global width value
®2	The current indentation level
®3	The current output column
®4	The current temporary indentation level

The registers can also be set by the following formatting commands:

=dn	Set register d to the value of expression n.

There are 10 registers, ®0 - ®9. These registers contain numeric values. Five of these registers belong to the formatter, the remainder to the user's format commands. The user registers are saved when a new context is entered and restored on exit. When a new context is entered, the user registers are set to the values of the outer context.

Numeric Expressions

```
expr  :   term
      |   expr + term
      |   expr - term
      ;

term  :   number
      |   ®d
      ;
```

Numeric expressions are sums and differences of terms and registers.

Formatting Commands

Formatting is done via yet another MPW language. The language is a prefix language; the operator is specified before the arguments to the operator. Blanks and tabs appearing between commands are ignored, however no blanks or tabs can appear between the operator and its arguments.

Indentation

The indent is where output lines will begin. The indent remains in effect until the closing of the context in which the indentation took place (think of the "{" and "}" surrounding compound statements).There is a global indent change value stored in register ®0 which can be used in expressions such as "i+®0" and "i-®0". The indent can also be set to the current column by using register ®3, as in "i®3". This is most useful for establishing the indent for lines following this one.

Because indent is preserved as part of a context, it is not necessary for glue to restore the previous indent. Indent is restored automatically when the end of a context is reached. A consequence of this is that it is not possible for a nested context to modify the indentation of an upper context. This is a minor limitation.

Permanent indent

i<n>	Set indent to absolute column n
i+<n>	Set indent to current indent + n
i-<n>	Set indent to current indent - n

Temporary indent

t<n>	Set temporary indent to absolute column n
t+<n>	Set temporary indent to current indent + n
t-<n>	Set temporary indent to current indent - n

The permanent indent applies to all lines output for the remainder of the current context. The temporary indent is used for continuation lines. It is the column output lines will begin when the line being written overflows the column width. It remains in effect until a newline, either from the source or from other formatting commands, occurs. At that point, indentation reverts to the current indentation level. Whenever the indent is modified, the temporary indent is set to the same value as the indent. The effect of this is that continuation lines line up under the current indent.

Indent to column

c<n>	Position to column <n>. Does not newline

Emits blanks and tabs until column <n> is reached. If already beyond that column, nothing is emitted. Indent to column overrides permanent and temporary indent for this one line of output.

White space
Inter token spaces

s<n>	Require <n> blanks. <n> defaults to 1
s#<n>	Desire <n> blanks. <n> defaults to 1

Blanks are desired in certain places and required in others. Blanks can be used to display "if (…" instead of "if(…". Required blanks are always displayed. Desired blanks will be removed to fit more items on the line if so required.

Line breaks

n<n>	Require <n> newlines. <n> defaults to 1
n#<n>	Desire <n> newlines. <n> defaults to 1
/	If a newline required, put one here.
/+<n>	As above, but override global temporary indent.

Newlines separate output lines. Required newlines will always be emitted when seen. Desired newlines will appear only after all comments from the current input source line have been emitted. Continuation newlines, denoted by "/", are emitted when the output line would be too long. If followed by "+<n>", the temporary indent for the first and following continuation lines will be <n> more than the current indent. Otherwise, the most recently established temporary indentation level is used.

Required white space

&n	Require beginning of line
&s	Require last token was a space/newline

The beginning of line token ("&n") requires that there must not be any output on the current line (with the exception of indentation).

Asserted white space

!n	Assert beginning of line
!s	Assert that last token was a space

Asserting beginning of line ("!n") is used by labels to indicate that a logical beginning of line condition exists.

Other formatting commands

Declaration formatting

dr<n>	Set declaration width to <n>, associate "*" and "&" with object
dl<n>	Set declaration width to <n>, associate "*" and "&" with type

The declaration width is the number of characters taken by the type of a declaration. The declaration might have operators such as "*" and "&". To maximize the declaration width (so that following declarations will be aligned to largest preceding declaration) use "dl0" or "dr0".

Conditional formatting

?n{…}{…}	At beginning of line
?s{…}{…}	Last token was a space or newline
?i{…}{…}	Last token was an identifier-like thing
?o{…}{…}	Last token was an operator
?'({…}{…}	Last token was a '('
?'){…}{…}	Last token was a ')'
?'{{…}{…}	Last token was a '{'
?'}{…}{…}	Last token was a '}'
?';{…}{…}	Last token was a ';'

If the condition is true, then the first set of formatting instructions are used, otherwise the second set of formatting instructions are used. Conditional formatting commands can be nested.

Display the token

•	Contextual reference to the current item

A place can have 0, 1, or 2 items to display. These items are referenced by the symbol "•" in the format string. When there are more than one item to display, the "•" work from left to right.

Places and their default formatting

This sections describes the places where glue can be applied, the wonderfully mnemonic option names for the places, and the default glue.

If statement

```
    • if                                -if0 "&n"
        • ( •                           -if1 "s# • i®3 !n"
            • )     •                   -if2 "•/"
            • ;     •                   -if3 "&n c®2+®0 • n#"
            • { •                       -if4 "?n{}{s#} • &n i+®0"
            • } •                       -if5 "&n c®2-®0 • &n"
            • } • else                  -if6 "&n c®2-®0 &n • s# !n"
            • ?                         -if7 "&n i+®0"

    • else • if                         -else1 "&n • s !n"
    • else • { •                    -else3 "?n{}{s#} • &n i+®0"
        • } •                           -else6 "&n c®2-®0 • n#"
    • else • ?                          -else4 "&n • n# i+®0"
```

Switch statement

```
    • switch                            -switch0 "&n"
        • ( •                           -switch1 "s# • i®3 !n"
            • { •                       -switch2 "?n{}{s#} • n# i+®0"
                • case •                -switch3 "&n c®2-®0 • &s"
                • default •             -switch4 "&n c®2-®0 •"
                • : •                   -switch5 "•n#"
            • } •                       -switch7 "c®2-®0 &n • n#"
```

For statement

```
    • for                           -for0 "&n"
        • ( •                           -for1 "s# • i®3 !n"
            • ;                         -for3 "!s • !n s#"
            • ) •                       -for4 "!s • /"
            • { •                       -for5 "?n{}{s#} • n# i+®0"
            • ; •                       -for6 "&n c®2+®0 • n#"
            • ?                         -for7 "&n i+®0"
            • } •                       -for8 "c®2-®0 &n • n#"
```

While statement

```
    • while                         -while0 "&n"
        • ( •                           -while1 "s# • i®3 !n"
            • { •                       -while2 "?n{}{s#} • n# i+®0"
            • ; •                       -while3 "&n c®2+®0 • n#"
            • ?                         -while4 "&n i+®0"
            • } •                       -while5 "c®2-®0 &n • n#"
        • ) •                           -while6 "•/"
```

```
Do statement
    • do                                -do0 "&n"
        • ;                                 -do1 "&n i+®0 • n#"
        • { •                               -do2 "?n{}{s#} • n# i+®0"
        • ?                                 -do3 "&n i+®0"
        • } •                               -do4 "c®2-®0 • ?n{}{s#} !n"
            • while                     -do5 "&n"
            • ( •                           -do6 "s# • i®3 !n"

Compound statement
    • { •                               -block1 "&n • i+®0 n#"
    • } •                               -block2 "&n c®2-®0 • n#"

Goto, break and return
    • goto •                                -goto1 "&n • &s"
    • break •                               -break1 "&n • !n"
    • return •                              -return1 "&n •& s"

Other statement
    • stmt                              -stmt0 "&n"
        • ; •                               -stmt1 "!s • n#"

Struct, union and class declarations
    • struct                                -struct0 ""
        • : •                               -struct1 "/ s# • i®3-1 s"
        • , •                               -struct2 "?n{•s#}{•/?n{}{s#}} !n"
            • { •                           -struct3 "?n{}{s#} • n# i+®0"
            • public • : •              -struct5 "&n c®2-®0 • c®3 • n#"
            • } •                           -struct6 "&n c®2-®0 • s#"
            • ; •                           -struct7 "!s c0 • n#3"

Function declaration
    • fundef
        • name                              -fundef2 "&n"
        • ( •                               -fundef3 "• i®3 !n"
            • , •                           -fundef4 "?n{• t®3-1 s#}{•/?n{}{s#}} !n"
        • ) •                               -fundef6 "•"
            • decl                          -fundef7 "&n c0 dr+"
            • : •                           -fundef8 "&n i+®0 t+2 • s"
                • { •                       -fundef13 "&n c®2+2 • n"
                • } •                       -fundef14 "&n c®2+2 • n"
            • , •                           -fundef10 "?n{• s#}{•/?n{s#}{&n c®2+2}}"
        • { •                               -fundef11 "&n • n i+®0"
        • } •                               -fundef12 "&n c®2-®0 • n#3"
```

Other declarations

- decl `-decl0 "&n dr"`
  - , • `-decl1 "?n{• s#}{• / ?n{}{s#}} !n"`
  - = • `-decl2 "s# • s# i®3 !n"`
  - (• `-decl3 "• i®3 !n"`
  - { • `-decl5 "&n • i®3 n#"`
    - { • `-decl6 "&n • i®3"`
    - } • `-decl7 "•"`
    - , • `-decl8 "?n{i®3•s#}{•/?n{}{s#}} !n"`
  - } • `-decl9 "•"`
  - ; • `-decl10 "!s • n#"`

Expression

- expr
  - , • `-expr1 "?n{•s#}{•/?n{}{s#}} !n"`
  - ? • `-expr2 "/?n{}{s#} • s#"`
  - : • `-expr3 "/?n{}{s#} • s#"`
  - op • `-expr4 "/ &s • &s"`
  - (• `-expr5 "/• i®3 !n"`
  - = • `-expr6 "&s • &s i®3"`
  -) • `-expr7 "?o{!s}{}•/"`
  - name • `-expr8 "?i{&s}{}"`

Function call

- funcall
  - (• `-funcall2 "• i®3 !n"`
  - , • `-funcall3 "?n{•s#i®3}{•/?n{}{s#}} !n"`
  -) • `-funcall4 "•/"`

Label

- label • : • `-label1 "&n c0 • c0 • c®2 !n"`

Name

- name `-name1 "?i{&s}{}"`

Command line options

By default, cdent reads from standard input and writes to standard output. It sets tabs to 4 spaces, indents by 4 spaces, sets temporary indent to +2 from the current indent, places end-of-line comments at column 48 and sets line length to 120. Solitary source new lines are ignored, but multiple newlines are passed through.

An input source file can be specified on the command line. Instead of standard output, another file can be named. Default formats can be overridden using the option names given in the “Places and their default formatting” section.

-o *file* Output file

The single argument is the name of an output file. The output file will be overwritten with the formatted source.

-p Progress
Emit progress information during the formatting. The progress information is written to standard error

-t *n* Set the tab width
The default tab width is 4. Sequences of 4 or more spaces will be replaced by a tab character. This option specifies how many spaces there are to a single tab character.

-indent *n* Default indentation
Lines within blocks and other statements are indented 4 more spaces than lines outside the block. This option specifies the indentation to use instead of 4.

-ll *n* Line length
Output lines are broken when they become longer than the line length. The default line length is 120.

-cc *n* Comment column
Comments appearing at the end of a line are aligned. The default column is 48. This option changes the comment column.

-allnl Preserve source newlines
Single newlines from the source are ignored except when long output lines are reformatted. Setting this option will preserve all source newlines in the output.

-nonl Ignore source newlines
Setting this option causes all source newlines to be ignored. The only newlines appearing in the output will those generated by the glue or by line overflows.

-trace *option* Debug cdent
Option is either *parse* or *formatting*. cdent will display cryptic lines of text during execution. Have fun.

Embedded formatting commands
Some formatting commands can be embedded in the source. These formatting commands are embedded in comments. If the first character following the comment lead-in is a "*ƒ*", the embedded formatting command follows it. At this writing, only one formatting command is available.

"Can't touch this" *ƒ*- and *ƒ*+
Frequently there are blocks of source text which should not be reformatted, either because the formatter will screw it up (sad but true) or because the tables are specially formatted data. To prevent this source from being formatted, the command "*ƒ*-" turns off formatting until the line following the command "*ƒ*+".

Appendix
How long output lines are formatted
Applying the default formats to most source files produces acceptably formatted output. However, some source files will cause the output line length to be exceeded. When this occurs, the formatter applies the following steps until the line fits:

1. Remove optional blanks.
The source is reformatted, with optional blanks (those whose glue is "s#") removed.

2. Use previously ignored source newlines.
Previously ignored newlines are inserted. This is done under the premise that the

source was not some bazoo test case but something relatively close to what was desired. Consequently, we re-use the newlines to break the output.

3. Insert a conditional newline.

If the re-use of source newlines doesn't help, conditional newlines (those whose glue is "/") are converted into real newlines. This is done after re-use of source newlines under the premise that source newlines are more likely correct than conditionally newlines.

4. Insert a newline.

Drop back and punt. A newline is inserted and formatting continues. This can yield very bizarre output on the right hand side of the affected lines. The output is valid, but squirrelly.

Treatment of newlines from the source

As noted above, source newlines are handled as special cases. Single newlines are ignored (except when reformatting long lines), but multiple newlines are retained. This allows the formatter to preserve blank lines following break statements inside switch statements if they occur without adding more special cases. It also allows the inclusion of blank lines separating declarations from the body of a block. It's the right thing to do.

Parsing and how it affects formatting

The formatter must parse the source. The parser is best described as a "slob" parser. Because cdent does not do file inclusion or macro expansion, it must be lenient in parsing. This leniency means that declarations will not always be recognized as declarations if the declaration begins with a typedef name. It also means that all code in any branch of a conditional must be valid; the technique of incorporating comments by preceding text with a "#if 0" will fail with a syntax error.