Subject: idl.el - IDL/PV-WAVE CL editing mode for GNU Emacs (long)
Posted by chase on Wed, 01 Sep 1993 21:33:34 GMT
View Forum Message <> Reply to Message

I have taken a long time to update my idl.el, the emacs editing mode for IDL and PV-WAVE CL programs. I also added many ideas from wave.el by Lubos Pochman of Precision Visuals, Boulder.

This is my first release version. I had given out a version I put together months ago to a few people. I would like feedback and bug reports so I can fix this up in a smoothly running machine.

I would appreciate if someone who is mantaining an IDL software archive would volunteer to put this in the archive. Let me know so I can send you updates.

I also have an Emacs mode (idl-shell-mode.el) for running IDL as a subprocess of Emacs. The advantage of this is that it lets you use all the emacs commands you are familiar with. It especially helps in debugging code, as it allows you to interactively set breakpoints and step through the code with simple keystrokes in the file being debugged with a pointer to the current line where IDL has stopped. It only works with GNU Emacs version 19. It is not polished, but I will give it away to anyone who wants to try it out and give me some feedback. Just ask.

Okay, here is idl.el. Place it somewhere where emacs will find it. See the Installation section. After installation, use `M-x idl-mode' to place a buffer in IDL mode. For help, in IDL mode use `C-h m'. Also look at the Customization variables section. Experiment around with the various keybindings (on a file you don't mind modifying.)

I hope this is satisfactory.

Cheers,
Chris Chase

```
-------- Cut Here -------------------
;;; idl.el --- IDL and WAVE CL editing mode for GNU Emacs
;;
;; Copyright (C) 1993  Chris Chase <chase@jackson.jhuapl.edu>
;;
;; Author: chase@jackson.jhuapl.edu
;; Keywords: languages
;; Version: 1.2 (1993/09/01 21:03:28)
;;
;; This file is not part of the GNU Emacs distribution but is
;; intended for use with GNU Emacs.
```

```
;;
;; 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 the GNU Emacs distribution; if not, write to the Free
;; Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139,
;; USA.

;;; Commentary:

;; In distant past, based on pascal.el. Though bears little
;; resemblance to that now.
;;
;; Incorporates many ideas, such as abbrevs, action routines, and
;; continuation line indenting, from wave.el.
;; wave.el original written by Lubos Pochman, Precision Visuals, Boulder.
;;
;; Installation:
;;
;;   To install put the following in your .emacs file (you may want
;;   to specify a complete path for the idl.el file if it is not in
;;   a directory contained in `load-path'):
;;
;;  (setq auto-mode-alist
;;      (append
;;       '(("\\.pro$" . idl-mode))
;;       auto-mode-alist))
;;   ;; Use a complete pathname in place of "idl" if necessary.
;;  (autoload 'idl-mode "idl"
;;   "Major mode for editing IDL/WAVE CL .pro files" t)
;;
;; Revision History:
;;
;; idl.el,v
;; Revision 1.2  1993/09/01  21:03:28  chase
;; Release version. Major rewrite, too many changes to
;; list. Incorporated ideas from wave.el, i.e., abbrevs, indent action
;; routines, continuation line indentation, etc.
;;
;;
```

```
;; Known problems:
;;
;;   Tabs and spaces are treated equally as whitespace when filling a
;;   comment paragraph.  To accomplish this, tabs are permanently
;;   replaced by spaces in the text surrounding the paragraph, which
;;   may be an undesirable side-effect.  Replacing tabs with spaces is
;;   limited to comments only and occurs only when a comment
;;   paragraph is filled via `idl-fill-paragraph'.
;;
;;   By using idl.el you should not need a tab other than perhaps
;;   the first line of a comment paragraph.
;;
;;   "&" is ignored when parsing statements.
;;   Avoid muti-statement lines (using "&") on block begin and end
;;   lines.  Multi-statement lines can mess up the formatting, for
;;   example, multiple end statements on a line: endif & endif.
;;   Using "&" outside of block begin/end lines should be okay.
;;
;;

;;;   Customization variables

;; For customized settings, change these variables in your .emacs file
;; using `idl-mode-hook'. For example:
;;
;;  (setq idl-mode-hook
;;  ;; For emacs version 19 replace above line with (add-hook 'idl-mode-hook
;; (function
;;  (lambda ()
;;    (setq
;;     idl-block-indent 3
;;     idl-main-block-indent 3
;;     idl-end-offset -3
;;     idl-continuation-indent 1
;;     idl-abbrev-change-case t
;;     idl-hang-indent-regexp ": "
;;     idl-auto-fill-mode 0  ; Turn off auto filling
;;     abbrev-mode 0   ; Turn off abbrevs
;;     ))))

;;; Variables for indentation behavior

(defvar idl-block-indent 4
  "*Extra indentation applied to block lines.")

(defvar idl-main-block-indent 0
  "*Extra indentation for the main block of code.
That is the block between the FUNCTION/PRO statement and the end
```

```
statement for that program unit.")

(defvar idl-end-offset -4
  "*Extra indentation applied to block end lines.
A value equal to negative `idl-block-indent' line
up end lines with the block begin lines.")

(defvar idl-continuation-indent 2
  "*Extra indentation applied to continuation lines.")

(defvar idl-newline-and-indent t
  "*Automatically indents current line, inserts newline and indents it.")

(defvar idl-hanging-indent t
  "*If set non-nil then comment paragraphs are indented under the
hanging indent given by `idl-hang-indent-regexp' match in the first line
of the paragraph.")

(defvar idl-hang-indent-regexp "- "
  "*Regular expression matching the position of the hanging indent
in the first line of a comment paragraph. The size of the indent
extends to the end of the match for the regular expression.")

;;; Variables for abbrev and action behavior

(defvar idl-surround-by-blank t
  "*If nil disables `idl-surround'.
By default, `=',`<',`>',`&',`,' are surrounded with spaces by `idl-surround'.
See help for `idl-indent-action-table' for symbols using `idl-surround'.

Also see the default key bindings for keys using `idl-surround'. Keys are
bound to `idl-surround' by binding them using `idl-pad'. See `idl-mode-map'
bindings below for examples.

See help for `idl-surround'.")

(defvar idl-show-block t
  "*If non-nil point blinks to block beginning for `idl-show-begin'.")

(defvar idl-abbrev-move nil
  "*If non-nil the abbrev hook can move point.
Set to nil by `idl-expand-region-abbrevs'. See the abbrev definitions,
`list-abbrevs', for abbrevs that move point. Moving point is useful, for
example, to place point between parentheses expanded functions.
See `idl-check-abbrev'.")

(defvar idl-reserved-word-upcase t
"*If non-nil, reserved words will be changed to upper case via abbrev expansion.
```

```
If nil case of reserved words is controlled by `idl-abbrev-change-case'.")

(defvar idl-abbrev-change-case nil
  "*If non-nil, then abbrevs will be made upper case.
If the value is `down then abbrevs will be forced to lower case.
If nil do not change the case of expansion.
Ignored for reserved words if `idl-reserved-word-upcase' is non-nil.")

;;; Types of comments

(defvar idl-line-comment ";;;\\|^;"
  "*Regular expression for a line comment. The indentation of this
type of comment will not be changed.")

(defvar idl-code-comment ";;[^;]"
  "*Regular expression for comment indented as code.")

(defvar idl-column-comment ";[^;]"
  "*Regular expression for a right margin comment.")

;;; Action Table.
;;
;; The average user may have difficulty modifying this.
;; If you want to try, modify it in your idl-mode-hook rather than here.
;; Try copying the entire statement to your hook, changinng the `defvar'
;; to a `setq' and deleting the documentation string. Then edit its value
;; at your own risk (see help on `idl-do-action' and `idl-surround').
;;
(defvar idl-indent-action-table
 '(("!". (capitalize-word 1));; Capitalize system vars
  ("&" . (idl-surround 1 1));; Add spaces to mulit stmt sign
  ("=" . idl-expand-equal);; Add spaces to equal sign if not keyword
  ("<" . (idl-surround 1 1));; Add spaces to less than
  (">" . (idl-surround 1 1));; Add spaces to greater than
  ("," . (idl-surround 0 1));; Add space after comma, remove space before
  ("\\<\\(pro\\|function\\)\\>[ \t]*\\<" . (capitalize-word 1));; Capitalize name
  ("\\<common\\>[ \t]*\\<" . (capitalize-word 1));; Capitalize name
        ("^[ \t]*\\<\\w*\\>:" . (capitalize-word -1));; Capitalize label
  )
  "*Associated array containing action lists of search string (car),
and function as a cdr. `idl-indent-line' function searches the table
and if match is found, executes the function. See documentation for
`idl-do-action' for a complete description of the action lists.")

;;; Documentation header and history keyword.

(defvar idl-file-header
  (list nil
"\;+
\; NAME:
\;
\;
\;
\; PURPOSE:
\;
\;
\;
\; CATEGORY:
\;
\;
\;
\; CALLING SEQUENCE:
\;
\;
\;
\; INPUTS:
\;
\;
\;
\; OPTIONAL INPUTS:
\;
\;
\;
\; KEYWORD PARAMETERS:
\;
\;
\;
\; OUTPUTS:
\;
\;
\;
\; OPTIONAL OUTPUTS:
\;
\;
\;
\; COMMON BLOCKS:
\;
\;
\;
\; SIDE EFFECTS:
\;
\;
\;
\; RESTRICTIONS:
\;
\;
```

```
\;
\; PROCEDURE:
\;
\;
\;
\; EXAMPLE:
\;
\;
\;
\; MODIFICATION HISTORY:
\;
\;-
")
  "*A list (PATHNAME STRING) specifying the doc-header template to use for
summarizing a file. If PATHNAME is non-nil then this file will be included.
Otherwise STRING is used. If NIL, the file summary will be omitted.
For example you might set PATHNAME to the path for the
lib_template.pro file included in the IDL distribution.")

(defvar idl-doc-modifications-keyword "HISTORY"
  "*The modifications keyword to use with the log documentation commands.
A ":" is added to the keyword end.
Inserted by doc-header and used to position logs by doc-modification.
If NIL it will not be inserted.")

;;; End customization variables section

;;; It is not likely that the USER would want to change any of the following.

(defvar idl-comment-line-start-skip "^[ \t]*;"
  "*Regexp to match the start of a full-line comment.
That is the _beginning_ of a line containing a comment delmiter `\;' preceded
only by whitespace.")

(defvar idl-begin-block-reg "\\<\\(pro\\|function\\|begin\\|case\\)\\>"
  "*Regular expression to find the beginning of a block. The case does
not matter. The search skips matches in comments.")

(defvar idl-begin-unit-reg "\\<\\(pro\\|function\\)\\>\\|\\`"
  "*Regular expression to find the beginning of a unit. The case does
not matter.")

(defvar idl-end-unit-reg "\\<\\(pro\\|function\\)\\>\\|\\'"
  "*Regular expression to find the line that indicates the end of unit.
This line is the end of buffer or the start of another unit. The case does
not matter. The search skips matches in comments.")

(defvar idl-continue-line-reg "\\<\\$"
  "*Regular expression to match a continued line.")

(defvar idl-end-block-reg
  "\\<end\\(\\|case\\|else\\|for\\|if\\|repeat\\|while\\)\\>"
  "*Regular expression to find the end of a block. The case does
not matter. The search skips matches found in comments.")

(defvar idl-fill-function (if (fboundp 'iconify-frame)
          'auto-fill-function
        'auto-fill-hook)
  "IDL mode auto fill function.
Value is auto-fill-hook for before emacs v19 or is auto-fill-function
for emacs v19 and later.")

(defvar idl-comment-indent-function (if (fboundp 'iconify-frame)
      'comment-indent-function
           'comment-indent-hook)
  "IDL mode comment indent function.
Value is comment-indent-hook for before emacs v19 or is
comment-indent-function for emacs v19 and later.")

;; Note that this is documented in the v18 manuals as being a string
;; of length one rather than a single character.
;; The code in this file accepts either format for compatibility.
(defvar idl-comment-indent-char ?
  "Character to be inserted for IDL comment indentation.
Normally a space.")

(defvar idl-continuation-char ?$
  "Character which is inserted as a last character on previous line by
   \\[idl-split-line] to begin a continuation line.  Normally $.")

(defvar idl-doclib-start "^;+\\+"
  "*Start of document library header.")

(defvar idl-doclib-end "^;+-"
  "*End of document library header.")

(defvar idl-startup-message t
  "*Non-nil displays a startup message when `idl-mode' is first called.")

(defvar idl-mode-version "0.6b")

(defmacro idl-pad (&rest args)
  "Creates an interactive function that calls `idl-surround' with arguments ARGS.
Useful for binding keys to `idl-surround' since the binding can not specify arguments
to the bound function."
  (` (function (lambda ()
   "Self insert and execute `idl-surround' with args for padding."
   (interactive)
   (insert last-command-char)
   (, (append '(idl-surround) args))))))

(defmacro idl-keyword-abbrev (&rest args)
  "Creates a function for abbrev hooks that calls `idl-check-abbrev' with args."
  (` (function (lambda ()
   (, (append '(idl-check-abbrev) args))))))

(defvar idl-mode-syntax-table nil
  "Syntax table in use in `idl-mode' buffers.")

(if idl-mode-syntax-table
    ()
  (setq idl-mode-syntax-table (make-syntax-table))
  (modify-syntax-entry ?+   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?-   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?*   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?/   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?^   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?#   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?=   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?%   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?<   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?>   "."  idl-mode-syntax-table)
  (modify-syntax-entry ?\'  "\"" idl-mode-syntax-table)
  (modify-syntax-entry ?\"  "\"" idl-mode-syntax-table)
  (modify-syntax-entry ?\\  "\\" idl-mode-syntax-table)
  (modify-syntax-entry ?_   "w"  idl-mode-syntax-table)
  (modify-syntax-entry ?{   "\(}"  idl-mode-syntax-table)
  (modify-syntax-entry ?}   "\){"  idl-mode-syntax-table)
  (modify-syntax-entry ?$   "w"  idl-mode-syntax-table)
;; `.' must have word syntax for abbrevs beginning with `.' to work.
  (modify-syntax-entry ?.   "w"  idl-mode-syntax-table)
  (modify-syntax-entry ?\;  "<"  idl-mode-syntax-table)
  (modify-syntax-entry ?\n  ">"  idl-mode-syntax-table)
  (modify-syntax-entry ?\f  ">"  idl-mode-syntax-table))

(defvar idl-mode-map ()
  "Keymap used in IDL mode.")

(defvar idl-debug-map nil
  "Keymap used in debugging in conjunction with `idl-shell-mode'.
It is set upon starting `idl-shell-mode'.")
(fset 'idl-debug-map (make-sparse-keymap))

(if idl-mode-map
    ()
  (setq idl-mode-map (make-sparse-keymap))
  (define-key idl-mode-map "'"        'idl-show-matching-quote)
  (define-key idl-mode-map "\""       'idl-show-matching-quote)
  (define-key idl-mode-map "&"        (idl-pad 1 1))
  (define-key idl-mode-map "="        'idl-equal)
  (define-key idl-mode-map "<"        (idl-pad 1 1))
  (define-key idl-mode-map ">"        (idl-pad 1 1))
  (define-key idl-mode-map ","        (idl-pad 0 1))
  (define-key idl-mode-map "\M-\t"    'idl-hard-tab)
  (define-key idl-mode-map "\C-c\t" 'idl-toggle-comment-region)
  (define-key idl-mode-map "\C-\M-a"  'idl-beginning-of-subprogram)
  (define-key idl-mode-map "\C-\M-e"  'idl-end-of-subprogram)
  (define-key idl-mode-map "\M-\C-h"  'idl-mark-subprogram)
  (define-key idl-mode-map "\M-\C-n"  'idl-forward-block)
  (define-key idl-mode-map "\M-\C-p"  'idl-backward-block)
  (define-key idl-mode-map "\M-\C-d"  'idl-down-block)
  (define-key idl-mode-map "\M-\C-u"  'idl-backward-up-block)
  (define-key idl-mode-map "\M-\r"    'idl-split-line)
  (define-key idl-mode-map "\M-\C-q"  'idl-indent-subprogram)
  (define-key idl-mode-map "\C-c\C-p" 'idl-previous-statement)
  (define-key idl-mode-map "\C-c\C-n" 'idl-next-statement)
  (define-key idl-mode-map "\r"       'idl-newline)
  (define-key idl-mode-map "\t"       'idl-indent-line)
  (define-key idl-mode-map "\C-c\C-a" 'idl-auto-fill-mode)
  (define-key idl-mode-map "\M-q" 'idl-fill-paragraph)
  (define-key idl-mode-map "\C-c\C-d"  'idl-doc-header)
  (define-key idl-mode-map "\C-c\C-h"  'idl-doc-modification)
  (define-key idl-mode-map "\C-c\C-c" 'idl-case)
  (define-key idl-mode-map "\C-c\C-f" 'idl-for)
  (define-key idl-mode-map "\C-x" 'idl-debug-map)
  ;;  (define-key idl-mode-map "\C-c\C-f" 'idl-function)
  ;;  (define-key idl-mode-map "\C-c\C-p" 'idl-procedure)
  (define-key idl-mode-map "\C-c\C-r" 'idl-repeat)
  (define-key idl-mode-map "\C-c\C-w" 'idl-while))
;;;
;;; Abbrev Section
;;;
;;; When expanding abbrevs and the abbrev hook moves backward, an extra
;;; space inserted (this is the space typed by the user to expanded
;;; the abbrev).
;;;
;;;

(defvar idl-mode-abbrev-table nil)
(if idl-mode-abbrev-table
    ()
  (define-abbrev-table 'idl-mode-abbrev-table ())
  (let ((abbrevs-changed nil))
    ;;
    ;; Keywords, system functions, conversion routines
    ;;
    (define-abbrev idl-mode-abbrev-table  ".b"   "begin" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".co"  "common" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".c"   "case of" (idl-keyword-abbrev 3 t))
    (define-abbrev idl-mode-abbrev-table  ".cb"  "byte()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cx"  "fix()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cl"  "long()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cf"  "float()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cs"  "string()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cc"  "complex()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".cd"  "double()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".d"   "do" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".e"   "else" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".ec"  "endcase" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".ee"  "endelse" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".ef"  "endfor" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".ei"  "endif else if" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".el"  "endif else" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".en"  "endif" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".er"  "endrep" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".ew"  "endwhile" 'idl-show-begin)
    (define-abbrev idl-mode-abbrev-table  ".f"   "for do" (idl-keyword-abbrev 3 t))
    (define-abbrev idl-mode-abbrev-table  ".fu"  "function" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".g"   "goto," (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".h"   "help," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".i"   "if" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".k"   "keyword_set()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".n"   "n_elements()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".on"  "on_error," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".oi"  "on_ioerror," (idl-keyword-abbrev 0 1))
    (define-abbrev idl-mode-abbrev-table  ".ow"  "openw," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".or"  "openr," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".ou"  "openu," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".pr"   "pro" (idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  ".p"  "print," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".pt"  "plot," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".r"   "repeat until" (idl-keyword-abbrev 6 t))
    (define-abbrev idl-mode-abbrev-table  ".re"  "read," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".rf"  "readf," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".ru"  "readu," (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".rt"  "return" (idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  ".sc"  "strcompress()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".sn"  "strlen()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".sl"  "strlowcase()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".su"  "strupcase()" (idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  ".sm"  "strmid()" (idl-keyword-abbrev 1))
```

```
   (define-abbrev idl-mode-abbrev-table  ".sp"  "strpos()" (idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  ".st"  "strput()" (idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  ".sr"  "strtrim()" (idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  ".t"   "then" (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table  ".u"   "until" (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table  ".wu"  "writeu," (idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  ".w"   "while do" (idl-keyword-abbrev 3 t))
   ;;
   ;; This section is reserved words only. (From IDL user manual)
   ;;
   (define-abbrev idl-mode-abbrev-table "and"      "and"       (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "begin"    "begin"     (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "case"     "case"      (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "common"   "common"    (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "do"       "do"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "else"     "else"      (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "end"      "end"       'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endcase"  "endcase"   'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endelse"  "endelse"   'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endfor"   "endfor"    'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endif"    "endif"     'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endrep"   "endrep"    'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endrepeat"   "endrepeat"    'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endwhi"   "endwhi"    'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "endwhile" "endwhile"  'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table "eq"       "eq"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "for"      "for"       (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "function" "function"  (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "ge"       "ge"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "goto"     "goto"      (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "gt"       "gt"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "if"       "if"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "le"       "le"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "lt"       "lt"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "mod"      "mod"       (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "ne"       "ne"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "not"      "not"       (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "of"       "of"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "on_ioerror" "on_ioerror" (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "or"       "or"        (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "pro"      "pro"       (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "repeat"   "repeat"    (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "then"     "then"      (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "until"    "until"     (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "while"    "while"     (idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table "xor"      "xor"       (idl-keyword-abbrev 0 t))))
;;
;;
```

```
;;
(defun idl-mode ()
  "Major mode for editing IDL and WAVE CL .pro files.

Indentation:
  By default current and next lines are indented, when RET is
  entered. This can be suppressed by setting `idl-newline-and-indent'
  to nil. TAB is used for explicit indentation.

  Code Indentation:
    Variable `idl-block-indent' specifies relative indent for
    block statements(begin|case...end),

    Variable `idl-continuation-indent' specifies relative indent for
    continuation lines.

    Continuation lines inside {}, [], (), are indented by
    `idl-continuation-indent' after opening parenthesis.

    Continuation lines in PRO, FUNCTION declarations are indented
    just after the prodcedure/function name.

    Labels are indented with the code unless they are on a line by
    themselves.

  Comment Indentation:
    Text in full line comments is indented to previous comment line
    text (if any), unless previous comment line is empty.
    Code line comment is indented to the value of `comment-column'.

    This mode handles comment paragraphs to a limited degree. A
    comment paragraph consists of consecutive lines containing the
    same comment leader (the whitespace at the beginning of the line
    plus comment delimiters). Additionally, blank comment lines will
    separate comment paragraphs. The indentation of a comment
    paragraph is given by first, the hanging indent or second, the
    minimum indentation of the paragraph lines after the first line. A
    hanging indent is specified by the presence of a string matching
    `idl-hang-indent-regexp' in the first line of the paragraph.

    Additionally, when in auto-fill mode comments are wrapped and
    indented according to the previous line indentation or hanging
    indent. Code lines are continued appropriately. To toggle the
    auto-fill mode use `idl-auto-fill-mode', \\[idl-auto-fill-mode],
    rather than the normal `auto-fill-mode' function.

\; Variable - this is a hanging indent. Text on following lines will
\;            be indented like this, past the hyphen and the following
\;          single space. \(Note that in auto fill mode that an
\;          automatic return on a line containing a hyphen will cause
\;          a hanging indent. if this happens in the middle of a
\;          paragraph where you don't want it, using
\;          idl-fill-paragraph, \\[idl-fill-paragraph], will re-fill the paragraph
\;          according to the hanging-indent in the first paragraph
\;          line.\) You can change the expression for the hanging
\;          indent, `idl-hang-indent-regexp'.

\;    Indentation will also automatically follow that of the
\;       of the previous line when you are in auto-fill mode
\;       like this.

Comments beginning with:
\;\;\; - a line comment by itself, indentation is not changed.
\;\; - a comment indented as code.
\; - a right margin comment.
These can be changed.

Comments at the beginning of the line are not indented.

Variables controlling indentation style and extra features:

 `idl-block-indent'
    Extra indentation within blocks.  (default 4)
 `idl-continuation-indent'
    Extra indentation within continuation lines.  (default 2)
 `idl-end-offset'
    Extra indentation applied to block end lines. (default -4)
 `idl-main-block-indent'
    Extra indentation for a units main block of code. That is the
    block between the FUNCTION/PRO statement and the end statement for
    that program unit. (default 0)
 `idl-newline-and-indent'
    If non-nil, automatically indents current line, inserts newline and
    indents it.   (default is t)
 `idl-surround-by-blank'
    Automatically surrounds '=','<','>' with blanks, appends blank to comma.
    (default is t)
 `idl-startup-message'
    Set to nil to inhibit message first time idl-mode is used.
 `idl-hanging-indent'
    If set non-nil to make hanging indents. (default t)

Other features:
  Use `M-x list-abbrevs' to display a list of built-in abbrevs for keywords.
  The case of reserved words and abbrevs is controlled by
  `idl-reserved-word-upcase' and `idl-abbrev-change-case'.
  A documentation header can be inserted at the beginning of the current
  program unit (pro, function or main) with `idl-doc-header',
  \\[idl-doc-header]. Change log entries can be added to the current
  program unit with `idl-doc-modification', \\[idl-doc-modification].

  Turning on IDL mode calls the value of the variable `idl-mode-hook'.

Command Table:

  Many control constructs, e.g., FOR and CASE, can be created
  by typing Control-C followed by Control applied to the first character of
  the construct. The block movement commands replace the list movement
  commands in the global and work analogously to the list commands.
  Use [\\idl-split-line] to continue or split a code or comment line.

\\{idl-mode-map}"
  (interactive)
  (kill-all-local-variables)
  (if idl-startup-message
      (message "Emacs IDL mode version %s." idl-mode-version))
  (setq idl-startup-message nil)
  (setq local-abbrev-table idl-mode-abbrev-table)
  (set-syntax-table idl-mode-syntax-table)
  (make-local-variable 'indent-line-function)
  (setq indent-line-function 'idl-indent-line)
  (make-local-variable idl-comment-indent-function)
  (set idl-comment-indent-function 'idl-comment-hook)
  (make-local-variable 'comment-start-skip)
  (setq comment-start-skip ";+[ \t]*")
  (make-local-variable 'comment-start)
  (setq comment-start ";")
  (make-local-variable 'require-final-newline)
  (setq require-final-newline t)
  (make-local-variable 'abbrev-all-caps)
  (setq abbrev-all-caps t)
  (make-local-variable 'indent-tabs-mode)
  (setq indent-tabs-mode nil)
  (use-local-map idl-mode-map)
  (setq mode-name "IDL")
  (setq major-mode 'idl-mode)
  (setq abbrev-mode t)

  (make-local-variable idl-fill-function)
  (set idl-fill-function 'idl-auto-fill)
  (setq comment-end "")
  (make-local-variable 'paragraph-start)
  (setq paragraph-start (concat "^[ \t\f]*$\\|" page-delimiter))
  (make-local-variable 'paragraph-separate)
  (setq paragraph-separate "^[ \t\f]*$\\|^[ \t]*;+[ \t]*$")
  (make-local-variable 'paragraph-ignore-fill-prefix)
  (setq paragraph-ignore-fill-prefix nil)
  (make-local-variable 'parse-sexp-ignore-comments)
  (setq parse-sexp-ignore-comments nil)

  (run-hooks 'idl-mode-hook))

;;
;;  Done with start up and initialization code.
;;  The remaining routines are the code formatting functions.
;;

(defun idl-hard-tab ()
  "Inserts TAB in buffer in current position."
  (interactive)
  (insert "\t"))

(defun idl-check-abbrev (arg &optional reserved)
  "Reverses abbrev expansion if in comment.
Argument ARG is the number of characters to move point
backward if `idl-abbrev-move' is non-nil.
If optional argument RESERVED is non-nil then the expansion
consists of reserved words, which will be capitalized if
`idl-reserved-word-upcase' is non-nil.
Returns non-nil if abbrev is left expanded."
  (if (idl-quoted)
      (progn (unexpand-abbrev)
      nil)
    (if (and reserved idl-reserved-word-upcase)
 (upcase-region last-abbrev-location (point))
      (if idl-force-keyword-downcase
   (downcase-region last-abbrev-location (point))))
    (if idl-abbrev-move (backward-char arg))
    t))



(defun idl-in-comment ()
  "Returns t if point is inside a comment, nil otherwise."
  (save-excursion
    (let ((here (point)))
      (and (idl-goto-comment) (> here (point))))))

(defun idl-goto-comment ()
  "Move to start of comment delimiter on current line.
Moves to end of line if there is no comment delimiter.
```

```
Ignores comment delimiters in strings.
Returns point if comment found and nil otherwise."
  (let ((eos (progn (end-of-line) (point)))
 found)
    ;; Look for first comment delimiter not in a string
    (beginning-of-line)
    (setq found (search-forward comment-start eos 'lim))
    (while (and found (idl-in-quote))
      (setq found (search-forward comment-start eos 'lim)))
    (and found (not (idl-in-quote))
  (progn
    (backward-char 1)
    (point)))))

(defun idl-show-matching-quote ()
  "Insert quote and show matching quote if this is end of a string."
  (interactive)
  (let ((bq (idl-in-quote))
 (inq last-command-char))
    (if (and bq (not (idl-in-comment)))
 (let ((delim (char-after bq)))
   (insert inq)
   (if (eq inq delim)
       (save-excursion
  (goto-char bq)
  (sit-for 1))))
      ;; Not the end of a string
      (insert inq))))

(defun idl-show-begin ()
  "Finds the start of current block and blinks to it for a second."
  (if (idl-check-abbrev 0 t)  ; All end statements are reserved words
    (if idl-show-block
 (save-excursion
   ;; Move inside current block
   (idl-beginning-of-statement)
   (idl-block-jump-out -1 'nomark)
;;; Future feature: implement check for correct match between block begin and end.
;;; Would require an alist of matches between begin and end types.
;;;      (beep)
;;;      (message "Warning: Block beginning does not match type!")
   (message " ")
   (sit-for 1)))))

(defun idl-surround (&optional before after)
  "Surround the character before point with blanks.
Optional arguments BEFORE and AFTER affect the behavior before and
after the previous character, respectively, if their values are:
```

```
nil   - do nothing.
c > 0 - exactly c spaces.
0     - no spaces.

If the character before point is inside a string or comment
or `idl-surround-by-blank' is nil then do nothing."
  (if (not (or (idl-quoted)
	       (not idl-surround-by-blank)))
      (progn
	(if (integerp before)
	    (progn
	      (backward-char 1)
	      (delete-horizontal-space)
	      (if (> before 0)
		  (insert-char ?  before))
	      (forward-char 1)))
	(if (integerp after)
	    (progn
	      (delete-horizontal-space)
	      (if (> after 0)
		  (insert-char ?  after)))))))


(defun idl-equal ()
  "Two cases: Assignment statement, and keyword assignment. Keyword
assignment will occur when function or procedure is declared, blanks
will be removed from both sides of the equal sign.  All other
conditions will be treated as standard assignment statements and the
equal sign will be surrounded by blanks."
  (interactive)
  (insert last-command-char)
  (idl-expand-equal))

(defun idl-newline ()
  "Inserts a newline and indents the current and previous line."
  (interactive)
  ;;
  ;; Handle unterminated single and double quotes
  ;; If not in a comment and in a string then insertion of a newline
  ;; will mean unbalanced quotes.
  ;;
  (if (and (not (idl-in-comment)) (idl-in-quote))
      (progn (beep)
	     (message "Warning: unbalanced quotes?")))
  (newline)
  (if idl-newline-and-indent
      ;;
      ;; The current line is being split, the cursor should be at the
      ;; beginning of the new line skipping the leading indentation.
      (progn
 (beginning-of-line 0)
 (idl-indent-line)
 (forward-line)
 (idl-indent-line))
    ))
;;
;;  Use global variable 'comment-column' to set parallel comment
;;
;; Modeled on lisp.el
;; Emacs Lisp and IDL (Wave CL) have identical comment syntax
(defun idl-comment-hook ()
  "Compute indent for the beginning of the IDL comment delimiter."
  (if (looking-at idl-line-comment)
      (current-column)
    (if (looking-at idl-code-comment)
 (let ((tem (idl-calculate-indent)))
   (if (listp tem) (car tem) tem))
      (skip-chars-backward " \t")
      (max (if (bolp) 0 (1+ (current-column)))
    comment-column))))

(defun idl-split-line ()
  "Continue line by breaking line at point and indent the lines.
For a code line insert continuation marker. If the line is a comment line
then the new line will contain a comment with the same indentation."
  (interactive)
  (if (not (idl-in-comment))
      ;; For code line add continuation
      (progn
 (if (idl-in-quote)
     (progn
       (beep)
       (message "Warning: continuation inside string!!")))
 (insert " " idl-continuation-char)))
  (idl-indent-line)
  (indent-new-comment-line))

(defun idl-beginning-of-subprogram ()
  "Moves point to the beginning of the current program unit."
  (interactive)
  (idl-find-key idl-begin-unit-reg -1))

(defun idl-end-of-subprogram ()
  "Moves point to the start of the next program unit."
  (interactive)
  (idl-end-of-statement)
  (idl-find-key idl-end-unit-reg 1))

(defun idl-mark-subprogram ()
  "Put mark at beginning of program, point at end.
The marks are pushed."
  (interactive)
  (idl-end-of-statement)
  (idl-beginning-of-subprogram)
  (push-mark)
  (idl-forward-block)
  (exchange-point-and-mark))

(defun idl-backward-up-block (&optional arg)
  "Move to beginning of enclosing block if prbefix ARG >= 0.
If prefix ARG < 0 then move forward to enclosing block end."
  (interactive "p")
  (idl-block-jump-out (- arg) 'nomark))

(defun idl-forward-block ()
  "Move across next nested block."
  (interactive)
  (if (idl-down-block 1)
      (idl-block-jump-out 1 'nomark)))

(defun idl-backward-block ()
  "Move backward across previous nested block."
  (interactive)
  (if (idl-down-block -1)
      (idl-block-jump-out -1 'nomark)))

(defun idl-down-block (&optional arg)
  "Go down a block.
With ARG: ARG >= 0 go forwards, ARG < 0 go backwards.
Returns non-nil if successfull."
  (interactive "p")
  (let (status)
    (if (< arg 0)
 ;; Backward
 (let ((eos (save-excursion
       (idl-block-jump-out -1 'nomark)
       (point))))
   (if (setq status (idl-find-key idl-end-block-reg -1 'nomark eos))
       (idl-beginning-of-statement)
     (message "No nested block before beginning of containing block.")))
      ;; Forward
      (let ((eos (save-excursion
     (idl-block-jump-out 1 'nomark)
     (point))))
 (if (setq status (idl-find-key idl-begin-block-reg 1 'nomark eos))
     (idl-end-of-statement)
   (message "No nested block before end of containing block."))))
    status))

(defun idl-mark-doclib ()
  "Put point at beginning of doc library header, mark at end.
The marks are pushed."
  (interactive)
  (let (beg
 (here (point)))
    (goto-char (point-max))
    (if (re-search-backward idl-doclib-start nil t)
 (progn (setq beg (progn (beginning-of-line) (point)))
        (if (re-search-forward idl-doclib-end nil t)
     (progn
       (forward-line 1)
       (push-mark beg)
       (exchange-point-and-mark))
   (message "Could not find end of doc library header.")))
      (message "Could not find doc library header start.")
      (goto-char here))))

(defun idl-beginning-of-statement ()
  "Move to beginning of the current statement.
Skips back past statement continuations.
Point is placed at the beginning of the line whether or not this is an
actual statement."
  (if (save-excursion (forward-line -1) (idl-is-continuation-line))
      (idl-previous-statement)
    (beginning-of-line)))

(defun idl-previous-statement ()
  "Moves point to beginning of the previous statement.
Returns t if the current line before moving is the beginning of
the first non-comment statement in the file, and nil otherwise."
  (interactive)
  (let (first-statement)
    (if (not (= (forward-line -1) 0))
 ;; first line in file
 t
      ;; skip blank lines, label lines, include lines and comment lines
      (while (and
       ;; The current statement is the first statement until we
       ;; reach another statement.
       (setq first-statement
      (or
       (looking-at idl-comment-line-start-skip)
       (looking-at "[ \t]*$")
       (looking-at "[ \t]*\\b[a-zA-Z]+[a-zA-Z0-9$_]*\\b:[ \t]*$")
       (looking-at "^@")))
       (= (forward-line -1) 0)))
      ;; skip continuation lines
      (while (and
       (save-excursion
  (forward-line -1)
  (idl-is-continuation-line))
       (= (forward-line -1) 0)))
      first-statement)))

(defun idl-end-of-statement ()
  "Moves point to the end of the current IDL statement.
If not in a statement just moves to end of line. Returns position."
  (interactive)
  (while (and (idl-is-continuation-line)
       (= (forward-line 1) 0)))
  (end-of-line) (point))

(defun idl-next-statement ()
  "Moves point to beginning of the next IDL statement.
 Returns t if that statement is the last
 non-comment IDL statement in the file, and nil otherwise."
  (interactive)
  (let (last-statement)
    (idl-end-of-statement)
    ;; skip blank lines, label lines, include lines and comment lines
    (while (and (= (forward-line 1) 0)
  ;; The current statement is the last statement until
  ;; we reach a new statement.
  (setq last-statement
        (or
         (looking-at idl-comment-line-start-skip)
         (looking-at "[ \t]*$")
         (looking-at "[ \t]*\\b[a-zA-Z]+[a-zA-Z0-9$_]*\\b:[ \t]*$")
         (looking-at "^@")))))
    last-statement))

(defun idl-expand-equal ()
  "Two cases: Assignment statement, and keyword assignment. Keyword
assignment will occur when function or procedure is declared or
called, blanks will be removed from both sides of the equal sign.  All
other conditions will be treated as standard assignment statements and
the equal sign will be surrounded by blanks."
  (if (save-excursion
 (idl-look-at "\\<function\\>\\|\\<pro\\>>" 'cont 'beg))
      (idl-surround 0 0)
    (idl-surround 1 1)))

(defun idl-indent-line ()
  "Indents current IDL line as code or as a comment."
  (interactive)
  ;; Move point out of left margin.
  (if (save-excursion
 (skip-chars-backward " \t")
 (bolp))
      (skip-chars-forward " \t"))
  ;;
  ;; Try to keep point at current relative position in line.  Abbrevs
  ;; probably will already have been expanded if this is done
  ;; interactively, so they will probably not mess us up.  However,
  ;; action routines that are applied after point can throw off the
  ;; location for point. There is not a good way to deal with this
  ;; other than applying all actions before point followed by actions
  ;; after point, so we will just live with where point ends up.
  ;;
  (let ((loc (- (point-max) (point))))
    (save-excursion
      (beginning-of-line)
      (if (looking-at idl-comment-line-start-skip)
   ;; Indentation for a comment line
   (progn
     (skip-chars-forward " \t")
     (idl-indent-to (idl-comment-hook)))
 (progn
   ;;
   ;; Before indenting, expand abbrevs and run action routines.
   ;;
   ;; Expand Abbrevs on the line
   ;;
   (let ((end-region (save-excursion (idl-goto-comment) (point))))
     (idl-expand-region-abbrevs (progn (beginning-of-line) (point))
     end-region))
   ;;
   ;; Now execute all action routines on the line
   ;;
   (mapcar 'idl-do-action idl-indent-action-table)
   ;; Indent for code line
   (if (or
        ;; a label line
        (looking-at "^\\b[a-zA-Z]+[a-zA-Z0-9$_]*\\b:[ \t]*$")
        ;; a batch command
        (looking-at "^[ \t]*@"))
       ;; leave flush left
       nil
     ;; indent the line
     (idl-indent-to (idl-calculate-indent)))
   ;; Adjust parallel comment
   (end-of-line)
   (if (idl-in-comment)
       (indent-for-comment)))))
    ;;
    ;; Don't leave point in left margin
    ;;
    (goto-char (- (point-max) loc))))

(defun idl-do-action (action)
  "Perform an action repeatedly on a line.
ACTION is a list (REG . FUNC).  REG is a regular expression.  FUNC is
either a function name to be called with `funcall' or a list to be
evaluated with `eval'.  The action performed by FUNC should leave after
the match for REG - otherwise an infinite loop may be entered."
  (let ((action-key (car action))
 (action-routine (cdr action)))
    (beginning-of-line)
    (while (idl-look-at action-key)
      (if (listp action-routine)
   (eval action-routine)
 (funcall action-routine)))))

(defun idl-indent-to (col)
  "Indent the current line to column COL.
Indents such that first non-whitespace character is at column COL."
  (save-excursion
    (beginning-of-line)
    (delete-horizontal-space)
    (indent-to col)))

(defun idl-indent-subprogram ()
  "Indents program unit which contains point."
  (interactive)
  (save-excursion
    (idl-end-of-statement)
    (idl-beginning-of-subprogram)
    (let ((beg (point)))
      (idl-forward-block)
      (message "Indenting subprogram...")
      (indent-region beg (point) nil))
    (message "Indenting subprogram...done.")))

(defun idl-calculate-indent ()
  "Return appropriate indentation for current line as IDL code."
  (save-excursion
    (beginning-of-line)
    (cond
     ;; Check for beginning of unit - main (beginning of buffer), pro, or
     ;; function
     ((idl-look-at idl-begin-unit-reg)
      0)
     ;; Check for continuation line
     ((save-excursion
 (and (= (forward-line -1) 0)
      (idl-is-continuation-line)))
      (idl-calculate-cont-indent))
     ;; calculate indent based on previous and current statements
     (t (let ((the-indent
        ;; calculate indent based on previous statement
        (save-excursion
   (cond
    ((idl-previous-statement)
     0)
    ;; Main block
    ((idl-look-at idl-begin-unit-reg)
     (+ (idl-current-indent) idl-main-block-indent))
    ;; Begin block
    ((idl-look-at idl-begin-block-reg)
     (+ (idl-current-indent) idl-block-indent))
    ((idl-look-at idl-end-block-reg)
     (- (idl-current-indent) idl-end-offset idl-block-indent))
    ((idl-current-indent))))))
   ;; adjust the indentation based on the current statement
   (cond
    ;; End block
    ((idl-look-at idl-end-block-reg)
     (+ the-indent idl-end-offset))
    (the-indent)))))))

;;
;; Parenthesses balacing/indent
;;

(defun idl-calculate-cont-indent ()
  "Calculates the IDL continuation indent column from the previous statement.
Note that here previous statement means the beginning of the current
statement if this statement is a continuation of the previous line.
Intervening comments or comments within the previous statement can
screw things up if the comments contain parentheses characters."
  (save-excursion
    (let* (open
    (end-reg (progn (beginning-of-line) (point)))
    (close-exp (progn (skip-chars-forward " \t") (looking-at "\\s)")))
    (beg-reg (progn (idl-previous-statement) (point))))
      ;;
      ;; If PRO or FUNCTION declaration indent after name, and first comma.
      ;;
      (if (idl-look-at "\\<\\(pro\\|function\\)\\>")
   (progn
     (forward-word 1)
     (if (looking-at "[ \t]*,[ \t]*")
  (goto-char (match-end 0)))
     (current-column))
 ;;
 ;; Not a PRO or FUNCTION
 ;;
 ;; Look for innermost unmatched open paren
 ;;
 (if (setq open (car (cdr (parse-partial-sexp beg-reg end-reg))))
     ;; Found innermost open paren.
     (progn
       (goto-char open)
   ;; Line up with next word unless this is a closing paren.
       (cond
        ;; This is a closed paren - line up under open paren.
        (close-exp
  (current-column))
        ;; Empty - just add regular indent. Take into account
        ;; the forward-char
        ((progn
    ;; Skip paren
    (forward-char 1)
    (looking-at "[ \t$]*$"))
  (+ (current-column) idl-continuation-indent -1))
        ;; Line up with first word
        ((progn
    (skip-chars-forward " \t")
    (current-column)))))
   ;; No unmatched open paren. Just a simple continuation.
   (goto-char beg-reg)
   (+ (idl-current-indent)
      ;; Make adjustments based on current line
      (cond
       ;; Else statement
       ((progn
   (goto-char end-reg)
   (skip-chars-forward " \t")
   (looking-at "else"))
        0)
       ;; Ordinary continuation
       (idl-continuation-indent)))))))
```

```
(defun idl-find-key (key-reg &optional dir nomark limit)
  "Move in direction of the optional second argument DIR to the
next keyword not contained in a comment or string and ocurring before
optional fourth argument LIMIT. DIR defaults to forward direction.  If
DIR is negative the search is backwards, otherwise, it is
forward. LIMIT defaults to the beginning or end of the buffer
according to the direction of the search. The keyword is given by the
regular expression argument KEY-REG.  The search is case insensitive.
Returns position if successful and nil otherwise.  If found
`push-mark' is executed unless the optional third argument NOMARK is
non-nil. If found, the point is left at the keyword beginning."
  (or dir (setq dir 0))
  (or limit (setq limit (cond ((>= dir 0) (point-max)) ((point-min)))))
  (let (found
 (case-fold-search t))
    (save-excursion
      (if (>= dir 0)
   (while (and (setq found (and
       (re-search-forward key-reg limit t)
       (match-beginning 0)))
        (idl-quoted)
        (not (eobp))))
 (while (and (setq found (and
     (re-search-backward key-reg limit t)
     (match-beginning 0)))
      (idl-quoted)
      (not (bobp))))))
    (if found (progn
  (if (not nomark) (push-mark))
  (goto-char found)))))


(defun idl-block-jump-out (&optional dir nomark)
  "When optional argument DIR is non-negative, move forward to end of
current block using the `idl-begin-block-reg' and `idl-end-block-reg' regular
expressions. When DIR is negative, move backwards to block beginning.
Recursively calls itself to skip over nested blocks. DIR defualts to
forward. Calls `push-mark' unless the optional argument NOMARK is
non-nil. Movement is limited by the start of program units because of
possibility of unbalanced blocks."
  (interactive "P")
  (or dir (setq dir 0))
  (let* ((here (point))
  (limit (cond ((>= dir 0) (point-max)) ((point-min))))
  (block-limit (cond ((>= dir 0) idl-begin-block-reg) (idl-end-block-reg)))
  found
  unit-start
  (block-reg (concat idl-begin-block-reg "\\|" idl-end-block-reg))
  (unit-limit
   (cond
    ((>= dir 0)
     (or
      (save-excursion
        (end-of-line)
        (idl-find-key idl-end-unit-reg dir t limit))
      limit))
    ((or
      (save-excursion
        (idl-find-key idl-begin-unit-reg dir t limit))
      limit)))))
    (if (>= dir 0) (end-of-line)) ;Make sure we are in current block
    (if (setq found (idl-find-key  block-reg dir t unit-limit))
 (while (and found (looking-at block-limit))
   (if (>= dir 0) (forward-word 1))
   (idl-block-jump-out dir t)
   (setq found (idl-find-key block-reg dir t unit-limit))))
    (if (not nomark) (push-mark here))
    (if (not found) (goto-char unit-limit)
      (if (>= dir 0) (forward-word 1)))))

(defun idl-current-indent ()
  "Return the column of the indentation of the current line.
Skips any whitespace. Returns 0 if the end-of-line follows the whitespace."
  (save-excursion
    (beginning-of-line)
    (skip-chars-forward " \t")
    ;; if we are at the end of blank line return 0
    (cond ((eolp) 0)
   ((current-column)))))

(defun idl-is-continuation-line ()
  "Tests if current line is continuation line."
  (save-excursion
    (idl-look-at "\\<\\$")))

(defun idl-look-at (regexp &optional cont beg)
  "Searches current line from current point for the regular expression
REGEXP. If optional argument CONT is non-nil, searches to the end of
the current statement. If optional arg BEG is non-nil, search starts
from the beginning of the current statement. Ignores matches that end
in a comment or inside a string expression. Returns point if
successful, nil otherwise.  This function produces unexpected results
if REGEXP contains quotes or a comment delimiter. The search is case
insensitive.  If successful leaves point after the match, otherwise,
does not move point."
```

```
  (let ((here (point))
 (case-fold-search t)
 (eos (if cont
  (save-excursion (idl-end-of-statement) (point))
        (save-excursion (end-of-line) (point))))
 found)
    (if beg (idl-beginning-of-statement))
    (while (and (setq found (re-search-forward regexp eos t))
  (idl-quoted)))
    (if (not found) (goto-char here))
    found))

(defun idl-fill-paragraph (&optional nohang)
  "Fills paragraphs in comments.
A paragraph is made up of all contiguous lines having the same comment
leader (the leading whitespace before the comment delimiter and the
coment delimiter).  In addition, paragraphs are separated by blank
comment lines. The indentation is given by the hanging indent of the
first line, otherwise by the minimum indentation of the lines after
the first line. The indentation of the first line does not change.
Does not effect code lines. Does not fill comments on the same line
with code.  The hanging indent is given by the end of the first match
matching `idl-hang-indent-regexp' on the paragraph's first line . If the
optional argument NOHANG is non-nil then the hanging indent is
ignored."
  (interactive "P")
  ;; check if this is a comment line
  (if (save-excursion
 (beginning-of-line)
 (skip-chars-forward " \t")
 (looking-at comment-start))
      (let
   ((indent 999)
    point
    pre
    diff
    fill-prefix-reg
    comment-leader
    hang
    start
    end)
 ;; Change tabs to spaces in the surrounding paragraph.
 ;; The surrounding paragraph will be the largest containing block of
 ;; contiguous comment lines. Thus, we may be changing tabs in
 ;; a much larger area than is needed, but this is the easiest
 ;; brute force way to do it.
 ;;
 ;; This has the undesirable side effect of replacing the tabs
 ;; permanently without the user's request or knowledge.
 (save-excursion
   (backward-paragraph)
   (setq start (point)))
 (save-excursion
   (forward-paragraph)
   (setq end (point)))
 (untabify start end)
 ;;
 (setq here (point))
 (beginning-of-line)
 (re-search-forward
  (concat "^[ \t]*" comment-start "+")
  (save-excursion (end-of-line) (point))
  t)
 ;; Get the comment leader on the line and its length
 (setq pre (current-column))
 (setq fill-prefix-reg
       (regexp-quote
        (setq fill-prefix
       (buffer-substring (save-excursion
      (beginning-of-line) (point))
           (point)))))
 (setq comment-leader fill-prefix)
 ;; Mark the beginning and end of the paragraph
 (backward-paragraph)
 (while (or (not (looking-at fill-prefix-reg))
     (looking-at paragraph-separate))
   (forward-line))
 (setq start (point))
 (forward-paragraph)
 (beginning-of-line)
 (if (or (not (looking-at fill-prefix-reg))
  (looking-at paragraph-separate))
     (forward-line -1))  ; Go to last line of paragraph
 (end-of-line)
 ;; if at end of buffer add a newline (need this because
 ;; fill-region needs END to be at the beginning of line after
 ;; the paragraph or it will add a line).
 (if (eobp)
     (progn (insert ?\n) (backward-char 1)))
 ;; Set END to the beginning of line after the paragraph
 ;; END is calculated as distance from end of buffer
 (setq end (- (point-max) (point) 1))
 ;;
 ;; Calculate the indentation for the paragraph.
 ;;
 ;; In the following while statements, after one iteration
 ;; point will be at the beginning of a line in which case
 ;; the while will not be executed for the
 ;; the first paragraph line and thus will not affect the
 ;; indentation.
 ;;
 ;; First check to see if indentation is based on hanging indent.
 (if (and (not nohang) idl-hanging-indent
   (setq hang
         (save-excursion
    (goto-char start)
    (idl-calc-hanging-indent))))
     ;; Adjust lines of paragraph by inserting spaces so that
     ;; each line's indent is at least as great as the hanging
     ;; indent. This is needed for fill-paragraph to work with
     ;; a fill-prefix.
     (progn
       (setq indent hang)
       (beginning-of-line)
       (while (> (point) start)
  (re-search-forward comment-start-skip
       (save-excursion (end-of-line) (point))
       t)
  (if (> (setq diff (- indent (current-column))) 0)
      (progn
        (if (>= here (point))
     ;; adjust the original location for the
     ;; inserted text.
     (setq here (+ here diff)))
        (insert (make-string diff ? ))))
  (forward-line -1))
       )
   ;; No hang. Instead find minimum indentation of paragraph
   ;; after first line.
   ;; For the following while statement, since START is at the
   ;; beginning of line and END is at the the end of line
   ;; point is greater than START at least once (which would
   ;; be the case for a single line paragraph).
   (while (> (point) start)
     (beginning-of-line)
     (setq indent
    (min indent
         (progn
    (re-search-forward
     comment-start-skip
     (save-excursion (end-of-line) (point))
     t)
    (current-column))))
     (forward-line -1)))
 (setq fill-prefix (concat fill-prefix
      (make-string (- indent pre)
            ? )))
 ;; Replace comment delimiters in first line with blanks so
 ;; that the indentation of that lines comment remains the same
 (goto-char start)
 (move-to-column pre)
 (subst-char-in-region start (point) ?\; ?  nil)
 ;; Try to keep point at its original place
 (goto-char here)
 ;; Fill the paragraph
 (save-excursion (fill-region start (- (point-max) end)))
 ;; Put back the comment delimiter in the first line
 (save-excursion (goto-char start)
   (move-to-column pre)
   (delete-region start (point))
   (insert-string comment-leader))
 ;; When we want the point at the beginning of the comment
 ;; body fill-region will put it at the beginning of the line.
 (if (bolp) (skip-chars-forward (concat " \t" comment-start)))
 (setq fill-prefix nil))))

(defun idl-calc-hanging-indent ()
  "Calculate the position of the hanging indent for the comment paragraph.
The hanging indent position is given by a match with the
`idl-hang-indent-regexp'.  If not found returns nil."
  (save-excursion
    (beginning-of-line)
    (if (re-search-forward
  idl-hang-indent-regexp
  (save-excursion (end-of-line) (point))
  t)
 (current-column))))

(defun idl-auto-fill ()
  "Called to break lines in auto fill mode.
Places a continuation character at the end of the line
if not in a comment."
  (let ((com (idl-in-comment)))
    (do-auto-fill)
    (save-excursion
      (forward-line 0)
      ;; Indent the split line
      (idl-indent-line))
    (if (not com)
 ;; Was a code line - add continuation character
 (progn
   (save-excursion
     (end-of-line 0)
     (insert " $"))
   ;; Although do-auto-fill (via indent-new-comment-line) calls
   ;; idl-indent-line for the new line, re-indent again
   ;; because of the addition of the continuation character.
   (idl-indent-line))
      ;; Comment line - make hanging indent
      (if idl-hanging-indent
   (let ((here (- (point-max) (point)))
  (indent
   (save-excursion
     (forward-line -1)
     (idl-calc-hanging-indent)))
  pos)
     (if indent
  (progn
    ;; Remove whitespace between comment delimiter and
    ;; text and insert spaces for appropriate indentation.
    (beginning-of-line)
    (re-search-forward
     comment-start-skip
     (save-excursion (end-of-line) (point))
     t)
    (setq pos (point))
    (skip-chars-backward " \t")
    (delete-region (point) pos)
    (insert (make-string
      (- indent (current-column)) ? ))
    (goto-char (- (point-max) here)))
       ))))))


(defun idl-auto-fill-mode (arg)
  "Toggle auto-fill mode for IDL mode.
With arg, turn auto-fill mode on iff arg is positive.
In auto-fill mode, inserting a space at a column beyond `fill-column'
automatically breaks the line at a previous space."
  (interactive "P")
  (prog1 (set idl-fill-function
       (if (if (null arg)
        (not (symbol-value idl-fill-function))
      (> (prefix-numeric-value arg) 0))
    'idl-auto-fill
  nil))
    ;; update mode-line
    (set-buffer-modified-p (buffer-modified-p))))

(defun idl-doc-header (&optional nomark )
  "Insert a documentation header at the beginning of the unit.
Inserts the value of the variable idl-file-header. Sets mark before
moving to do insertion unless the optional prefix argument NOMARK
is non-nill."
  (interactive "P")
  (or nomark (push-mark))
  ;; make sure we catch the current line if it begins the unit
  (end-of-line)
  (idl-beginning-of-subprogram)
  (beginning-of-line)
  ;; skip function or procedure line
  (if (idl-look-at "\\<\\(pro\\|function\\)\\>")
      (progn
 (idl-end-of-statement)
 (if (> (forward-line 1) 0) (insert "\n"))))
  (if idl-file-header
      (cond ((car idl-file-header)
      (insert-file (car idl-file-header)))
     ((stringp (car (cdr idl-file-header)))
      (insert (car (cdr idl-file-header)))))))


(defun idl-doc-modification ()
  "Insert a brief modification log at the beginning of the current program.
Looks for an occurrence of the value of user variable
`idl-doc-modifications-keyword' if non-nil. Inserts time and user name
and places the point for the user to add a log. Before moving, saves
location on mark ring so that the user can return to previous point."
  (interactive)
  (push-mark)
  ;; make sure we catch the current line if it begins the unit
  (end-of-line)
  (idl-beginning-of-subprogram)
  (let ((pro (idl-look-at "\\<\\(function\\|pro\\)\\>"))
 (case-fold-search nil))
    (if (re-search-forward
  (concat idl-doc-modifications-keyword ":")
  ;; set search limit at next unit beginning
  (save-excursion (idl-end-of-subprogram) (point))
  t)
 (end-of-line)
      ;; keyword not present, insert keyword
      (if pro (idl-next-statement)) ; skip past pro or function statement
      (beginning-of-line)
      (insert "\n" comment-start "\n")
      (forward-line -2)
      (insert comment-start " " idl-doc-modifications-keyword ":")))
  (idl-newline)
  (beginning-of-line)
  (insert ";\n;\t" (current-time-string))
  (insert ", " (user-full-name))
  (insert " <" (user-login-name) "@" (system-name) ">")
  ;; Remove extra spaces from line
  (idl-fill-paragraph)
  ;; Insert a blank comment line to separate from the date entry -
  ;; will keep the entry from flowing onto date line if re-filled.
  (insert "\n;\n;\t\t"))

;;; CJC 3/16/93
;;; Interface to expand-region-abbrevs which did not work when the
;;; abbrev hook associated with an abbrev moves point backwards
;;; after abbrev expansion, e.g., as with the abbrev '.n'.
;;; The original would enter an infinite loop in attempting to expand
;;; .n (it would continually expand and unexpand the abbrev without expanding
;;; because the point would keep going back to the beginning of the
;;; abbrev instead of to the end of the abbrev). We now keep the
;;; abbrev hook from moving backwards.
;;;
(defun idl-expand-region-abbrevs (start end)
  "Expand each abbrev occurrence in the region.
Calling from a program, arguments are START END."
  (interactive "r")
  (save-excursion
    (goto-char (min start end))
    (let ((idl-show-block nil)  ;Do not blink
     (idl-abbrev-move nil)) ;Do not move
      (expand-region-abbrevs start end 'noquery))))

(defun idl-quoted ()
  "Returns t if point is in a comment or quoted string.
nil otherwise."
  (or (idl-in-comment) (idl-in-quote)))

(defun idl-in-quote ()
  "Returns location of the opening quote
if point is in a IDL string constant, nil otherwise.
Ignores comment delimiters on the current line.
Properly handles nested quotation marks and octal
constants - a double quote followed by an octal digit."
;;; Treat an octal inside an apostrophe to be a normal string. Treat a
;;; double quote followed by an octal digit to be an octal constant
;;; rather than a string. Therefore, there is no terminating double
;;; quote.
  (save-excursion
    ;; Because single and double quotes can quote each other we must
    ;; search for the string start from the beginning of line.
    (let* ((start (point))
    (eol (progn (end-of-line) (point)))
    (bq (progn (beginning-of-line) (point)))
    (endq (point))
    delim
    found)
      (while  (< endq start)
 ;; Find string start
 ;; Don't find an octal constant beginning with a double quote
 (if (re-search-forward "\"[^0-7]\\|'\\|\"$" eol 'lim)
     ;; Find the string end.
     ;; In IDL, two consecutive delimiters after the start of a
     ;; string act as an
     ;; escape for the delimiter in the string.
     ;; Two consecutive delimiters alone (i.e., not after the
     ;; start of a string) is the the null string.
     (progn
       ;; Move to position after quote
       (goto-char (1+ (match-beginning 0)))
       (setq bq (1- (point)))
       ;; Get the string delimiter
       (setq delim (char-to-string (preceding-char)))
       ;; Check for null string
       (if (looking-at delim)
    (progn (setq endq (point)) (forward-char 1))
  ;; Look for next unpaired delimiter
  (setq found (search-forward delim eol 'lim))
  (while (looking-at delim)
    (forward-char 1)
    (setq found (search-forward delim eol 'lim)))
  (if found
      (setq endq (- (point) 1))
    (setq endq (point)))
  ))
   (progn (setq bq (point)) (setq endq (point)))))
      ;; return string beginning position or nil
      (if (> start bq) bq)
      )))

(defun idl-case ()
  "Build skeleton case statment, prompting for the <expression>."
  (interactive)
  (insert "case ")
  (let ((selector (read-string "Selector Expression: ")))
    (progn
      (insert selector " of")
      (idl-newline)
      (idl-newline)
      (insert "endcase; case " selector )
      (idl-indent-line)))
  (end-of-line 0)
  (idl-indent-line))

(defun idl-for ()
  "Build skeleton for loop statment, prompting for the loop parameters."
  (interactive)
  (insert "for ")
  (insert (read-string "Loop expr: ") " do begin")
  (idl-newline)
  (idl-indent-line)
  (idl-newline)
  (insert "endfor")
  (idl-indent-line)
  (end-of-line 0)
  (idl-indent-line))


(defun idl-procedure ()
  (interactive)
  (insert "pro ")
  (let ((name (read-string "Procedure name: " )))
    (insert name )
    (idl-newline)
    (idl-newline)
    (insert "return ")
    (idl-newline)
    (insert "end; ")
    (insert name)
    (idl-indent-line)
    (end-of-line -2)))

(defun idl-function ()
  (interactive)
  (insert "function ")
  (let ((name (read-string "Function name: " )))
    (insert name )
    (idl-newline)
    (idl-newline)
    (insert "return ")
    (idl-newline)
    (insert "end; ")
    (insert name)
    (idl-indent-line)
    (end-of-line -2)))

(defun idl-repeat ()
  (interactive)
  (insert "repeat begin")
  (idl-newline)
  (idl-newline)
  (insert "endrep until ")
  (insert (read-string "Exit cond: ") )
  (idl-indent-line)
  (end-of-line 0)
  (idl-indent-line))

(defun idl-while ()
  (interactive)
  (insert "while ")
  (insert (read-string "Entry cond: "))
  (insert " do begin")
  (idl-newline)
  (idl-newline)
  (insert "endwhile")
  (idl-indent-line)
  (end-of-line 0)
  (idl-indent-line))

(defun idl-toggle-comment-region (beg end &optional n)
  "Comment the lines in the region if the first non-blank line is
commented, and conversely, uncomment region. If optional prefix arg
N is non-nil, then for N positive, add N comment delimiters or for N
negative, remove N comment delimiters.
Uses `comment-region' which does not place comment delimiters on
blank lines."
  (interactive "r\nP")
  (if n
      (comment-region beg end (prefix-numeric-value n))
    (save-excursion
      (goto-char beg)
      (beginning-of-line)
      ;; skip blank lines
      (skip-chars-forward " \t\n")
      (if (looking-at (concat "[ \t]*\\(" comment-start "+\\)"))
   (comment-region beg end
     (- (length (buffer-substring
         (match-beginning 1)
         (match-end 1)))))
 (comment-region beg end)))))

(provide 'idl)

;;; idl.el ends here
--
```

================================
Bldg 24-E188
The Applied Physics Laboratory
The Johns Hopkins University
(301)953-6000 x8529