Subject: Re: Emacs
Posted by J.D. Smith on Thu, 17 Jul 1997 07:00:00 GMT
View Forum Message <> Reply to Message

I have modified idl.el to work with IDL 5's class method definitions. Specifically, I changed to font-lock regexp for finding function/pro names, and redid the defun idl-unit-name for the same purpose (for imenu). I do not program in lisp, and this is the first bit of emacs lisp I have ever done (other than .emacs files), so I'd expect that there's a better way to get the job done. But it seems to work for me, and you might be interested in putting it on the ftp sight.

Thanks,

J.D. Smith

P.S. If modifications are made to my fixes, I'd appreciate a note letting me know.

**********BEGIN idl.el v 1.39**********
;;; idl.el --- IDL and WAVE CL editing mode for GNU Emacs
;; Author: chris.chase@jhuapl.edu
;; Maintainer: williams@irc.chmcc.org
;; Keywords: languages
;; Version: $Revision: 1.39 $ ($Date: 1997/07/16 16:19:00 $)
;;
;; LCD Archive Entry:
;; idl|Phil Williams|williams@irc.chmcc.org|
;; Editing Mode for IDL and PV-Wave procedure files.|
;; $Date: 1997/07/16 16:19:00 $|$Revision: 1.39 $|~/misc/idl.el|
;;
;; 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.
;;
;; See the mode description ("C-h m" in idl-mode or "C-h f idl-mode")
;; for features, key bindings, and info.
;;
;; Installation:
;;
;;   Should byte compile properly if you desire to do so (within emacs
;;   just execute "M-x byte-compile-file" followed by the path to the
;;   idl.el file.)
;;
;;   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)
;;
;; Source:
;;
;;   The latest version can be obtained via anonymous FTP at:
;;
;;   ftp://scuttle.chmcc.org/pub/idl_emacs/idl.el
;;
;; Acknowledgments:
;;
;;   Thanks to the following people for their contributions and
comments:
;;     Chris Chase <chris.chase@jhuapl.edu>    The Author
;;     Lubos Pochman <lubos@rsinc.com>
;;     sterne@dublin.llnl.gov (Phil)
;;     David Huenemoerder <dph@space.mit.edu>
;;     Patrick M. Ryan <pat@jaameri.gsfc.nasa.gov>
```

```
;;     (Xuyong Liu) <liu@stsci.edu>
;;     ryba@ll.mit.edu (Marty Ryba)
;;     Phil Williams <williams@irc.chmcc.org>
;;     Ulrik Dickow <dickow@nbi.dk>
;;     mugnier@onera.fr (Laurent MUGNIER)
;;     Stein Vidar H. Haugan <s.v.h.haugan@astro.uio.no>
;;     Simon Marshall <Simon.Marshall@esrin.esa.it>
;;
;; Revision History:
;;
;; $Log: idl.el,v $
;; Revision 1.39  1997/07/16  16:19:00 smith, J.D.
;; (idl-unit-name):  Hacked to recognize Class methods (Class::Method)
;; Amended Font-Lock function/pro name regexp for same purpose.
;;
;; Revision 1.38  1997/01/30  18:52:06  williams
;; Bookkeeping update.  New ftp and maintainer.
;;
;; Revision 1.37  1997/01/30 00:56:41  chase
;; (idl-abbrev-start-char): Allow the user to select the character that
;; start abbreviations in abbrev mode.  Previously '.'  was used which
;; caused a problem with structure fields acting as abbrevs.  Now
;; defaults to '\'.  Changes to this in idl-mode-hook will have no
;; effect.  Instead a user must set it directly using `setq' in the
;; .emacs file before idl.el is loaded.
;; (idl-calculate-cont-indent): Use forward-sexp instead of forward-word
;; to skip beyond an identifier.
;; (idl-mode): Added a check for availability for easy-menu.
;;
;; Revision 1.36  1996/11/20 16:59:32  chase
;; (idl-show-begin-check): Before calling `idl-show-begin' ensure that
an
;; END word is indeed a token and not the last part of a larger token,
;; i.e. foo_end.
;;
;; Revision 1.35  1996/11/15 19:44:25  chase
;; (idl-font-lock-defaults): change "." to word syntax to keep from
;; hilighting @file.pro statements and keywords used as IDL structure
;; tags.
;;
;; Revision 1.34  1996/11/12 22:34:30  chase
;; (idl-is-continuation-line): Fixed screw up in the undoing of "\\>" to
;; "\\S_" edits made previously.
;;
;; Revision 1.33  1996/11/12 21:27:38  chase
;; (idl-find-symbol-syntax-table): Added to handle "$_" as symbol syntax
;; in idl-find-key and idl-look-at.  This effectively gives the original
;; symbol searching behavior throughout idl.el code.
```

;; (idl-look-at, idl-find-key): Use idl-find-symbol-syntax-table.  These
;; functions should be used inside idl.el for finding symbols instead of
;; using "\\S_" in regular expressions.
;;
;; Revision 1.32  1996/11/11 17:38:53  chase
;; (idl-is-continuation-line): fixed for "$" as symbol syntax instead of
;; word syntax.
;;
;; Revision 1.31  1996/11/07 17:53:45  chase
;; (idl-mode-syntax-table): Changed the syntax of "$" and "_" to be
;; symbol constituent characters which means they are no longer
;; considered word constituent characters.  This changes how word
;; oriented commands work, e.g. M-d and M-DEL will no longer kill an
;; entire IDL identifier (variable name) containing "$" or "_".  To
;; operate on an entire IDL identifier (variable name) use the sexp
style
;; commands, e.g. backward-kill-sexp, mark-sexp, kill-sexp.
;; (idl-make-tags): Emacs tag creation uses new functions
;; idl-replace-string and idl-split-string.
;; (idl-doc-modification): Now uses idl-timestamp-hook which by default
;; uses idl-default-insert-timestamp.
;;
;; Revision 1.30  1996/09/17 15:43:01  chase
;; (idl-calc-hanging-indent): Allow matching last hanging indent regexp
;; on line via idl-use-last-hang-indent.
;; (idl-fill-paragraph): Leave untouched whitespace within indent length
;; in first line of filled paragraph.
;;
;; Revision 1.29  1996/08/29 17:38:29  chase
;; (idl-auto-fill):
;; (idl-mode):
;; (idl-fill-paragraph): Fixed paragraph filling bug that manifested
;; itself in newer versions of emacs.
;; (idl-mode): Set comment-multi-line to nil for filling to work in
;; Xemacs.
;; (idl-mode): fixes for easy-menu under XEmacs.
;; (idl-surround): Avoid clobbering indent when positioned at beginning
;; of text, but after indent.
;;
;; Revision 1.28  1995/12/01  20:33:52  chase
;; More font-lock changes from users.
;;
;; Revision 1.27  1995/09/25  15:39:07  chase
;; (idl-comment-hook): Use idl-begin-line-comment.
;; (idl-begin-line-comment): Allows customization of comment types
;; anchored at the beginning of line.
;;
;; Revision 1.26  1995/09/22  20:41:15  chase

;; (idl-code-abbrev): Added.  Unexpand quoted abbrevs, otherwise
optional
;; args are evaluated as a single list.
;; (idl-indent-line): Removed abbrev expansion for line which was
causing
;; other undesirable problems.  Can expand manually using
;; idl-expand-region-abbrevs.
;; (idl-surround): Fixed bug and endless loop problem when indenting.
;; Must move to end of whitespace after making space 'before' point.
;;
;; Revision 1.25  1995/09/14  13:09:41  chase
;; (idl-font-lock-keywords): Changed to ignore case.
;; (idl-mode): Choice of font-lock or hilit19 is now left to the user.
;; No hilighting is used by default.  Added appropriate instructions.
;; (idl-use-font-lock): Removed in favor of standard method for turning
;; on font-lock.
;; (idl-running-lemacs): Removed.
;; (idl-font-lock-keywords): Changed from defconst to defvar.
;; (idl-mode): Removed idl-fontify and put its functionality into
;; idl-mode.
;;
;; Revision 1.24  1995/07/20  23:23:07  chase
;; (idl-auto-fill): Need to check to see if filling actually needs to be
;; done (as of GNU Emacs v19.29).
;;
;; Revision 1.23  1995/07/12  13:55:31  chase
;; Fixed bug in usage of easy-menu-define - previous fix was lost
;; somehow.
;;
;; Revision 1.22  1995/07/07  22:16:56  chase
;; (idl-indent-line): Put back in abbrev expansion, but tries to handle
;; problem with point not staying at the end of an abbrev after
;; expansion.
;; (idl-elif): Added.
;; (idl-template): Open line if no newline after point.  This will catch
;; the end of buffer in addition to additional text on the line.
;;
;; Revision 1.21  1995/06/14  15:29:20  chase
;; Put capitalized keywords back in for font-lock.
;;
;; Revision 1.20  1995/06/12  16:24:49  chase
;; Added font-lock modifications from Phil Williams.
;; (idl-mode): Turn off hilit-auto-rehighlight when using font-lock
mode.
;; (idl-font-lock-keywords): Added.
;; (idl-use-font-lock): Added.
;; (idl-running-lemacs): Added.
;; (idl-fontify): Added.

;;
;; Revision 1.19  1995/06/09  13:52:24  chase
;; Fixed bug in usage of easy-menu-define.
;;
;; Revision 1.18  1995/01/26  16:58:52  chase
;; Added LCD Archive Entry.
;;
;; Revision 1.17  1995/01/25  16:41:49  chase
;; (idl-statement-type): Move past continuations.
;; (idl-auto-fill): Added option to fill only comment lines.
;; (idl-fill-comment-line-only): Added.  By default only fill comment
;; lines.
;; (idl-make-space): Place point abs(n) spaces from the left rather than
;; at the end of the whitespace.
;;
;; Revision 1.16  1994/10/26  21:57:51  chase
;; (idl-start-of-substatement): Added.
;; (idl-expand-equal): Can distinguish between keywords assignments and
;; assignment statements.
;; (idl-pad-keyword): Added for surrounding keyword '='.
;; (idl-calculate-cont-indent): case-fold-search set to t.
;; (idl-block-jump-out): case-fold-search set to t.
;; (idl-action-and-binding): Only set a binding if KEY has length 1
;; unless SELECT is 'both or 'noaction.
;; (idl-statement-match): Swapped key and values and and added
expression
;; to match next statement.
;; (idl-skip-label): Added.
;; (idl-indent-line): Optional keyword NOEXPAND.
;; (idl-indent-expand-table): Added for actions that are performed only
;; with explicit indentation.
;; (idl-indent-and-action): Added for the `indent-expand-table'.
;; (idl-action-and-binding): Uses `idl-indent-expand-table'.
;; (idl-statement-type): Determines the current statment type.
;; (idl-expand-equal): Uses `idl-statement-type' to determine if "=" is
;; for an assignment statement.
;; (idl-statement-match): Table for statement types.
;; (idl-label, idl-identifier, idl-variable, idl-sysvar): Added.
;; (idl-indent-action-table): Removed the default actions.  Users must
;; set their own.
;; (idl-split-line): Was continuing as a comment line when ';' was
inside
;; a string.
;; (idl-mode): Version 1.11 of imenu.el changed the names of certain
;; functions.  Added check.
;;
;; Revision 1.15  1994/06/29  22:46:03  chase
;; (idl-indent-left-margin): Replaces use of indent-to.  Inserts space

```
;; before markers at point.
;; (idl-indent-line): Use a marker to keep position of point.  Removed
;; expanding abbrevs on line - buggy.
;;
;; Revision 1.14  1994/06/29  01:34:26  chase
;; New FTP sites.
;;
;; Revision 1.13  1994/06/24  23:21:49  chase
;; (idl-mode-menus): new.  Added corrections for using easymenu.el.
;;
;; Revision 1.12  1994/06/22  23:10:49  chase
;; (idl-auto-fill): Fixed auto-filling when in a non-line comment and
;; when the whole comment is moved to the new line.
;; (idl-split-line): Split the line before indenting the split line.
;;
;; Revision 1.11  1994/05/13  16:24:52  chase
;; Remove ENDREPEAT as a end block reserved word.
;;
;; Revision 1.10  1994/03/30  17:56:32  chase
;; (idl-indent-line): Check for abbrev-mode before expanding abbrevs.
;; (idl-mark-statement): Added.
;; (idl-mode): Added hilit pattern setup.
;; (idl-hilit-patterns): Added for use with hilit19.el.
;;
;; Revision 1.9  1994/03/23  18:33:31  chase
;; Set imenu-create-index-function explicitly for imenu.el (the default
;; may be overridden) and load imenu as necessary.
;;
;; Revision 1.8  1994/03/21  22:52:13  chase
;; Cleaned up mode documentation.
;; Added menus using easymenu.el.
;; (idl-mark-block): Added.
;; (idl-push-mark): Added for compatibility btw Emacs 18/19.
;; Changed the binding of TAB to the default which indents. Changed \C-j
;; to be `newline-and-indent' and removed idl-newline from RET.  This
;; makes idl.el more like other programming modes in Emacs.  Changed the
;; defaults for "electric" padded keys (e.g. "=") to use a minimum
rather
;; than exact number of spaces.  Electric padding is off by default.
;; (idl-newline): Removed the variable idl-newline-and-indent.  The
;; user can avoid the "electric" newline by removing the binding.
;;
;;
;; Known problems:
;;
;;   "." caused problems with abbrevs being used with IDL structure
;;   fields.  So abbrevs were changed to start with "\".  A better fix
;;   would use the expand.el package instead of the built-in abbrev
```

```
;;   facility.  expand.el would also fix the next problem.
;;
;;   Moving the point backwards in conjunction with abbrev expansion
;;   does not work as I would like it, but this is a problem with
;;   emacs abbrev expansion done by the self-insert-command.  It ends
;;   up inserting the character that expanded the abbrev after moving
;;   point backward, e.g., ".cl" expanded with a space becomes
;;   "LONG( )" with point before the close paren.  I don't have a way
;;   around this except for a kludge in emacs version 19 for which I
;;   use `post-command-hook'.
;;
;;   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'.  Here is an example of what to place in your
.emacs.
;;
;;  (add-hook 'idl-mode-hook
;; ;; For emacs version 18 replace above line with (setq idl-mode-hook
;; (function
;;  (lambda ()
;;    (setq   ; Set options here
;;     idl-block-indent 3          ; Indentation settings
;;     idl-main-block-indent 3
;;     idl-end-offset -3
;;     idl-continuation-indent 1
;;          ; Leave ";" but not ";;" anchored at start of line.
;;          idl-begin-line-comment "^\;[^\;]"
;;          idl-surround-by-blank t     ; Turn on padding symbols =,<,>,
etc.
;;     abbrev-mode 1  ; Turn on abbrevs (-1 for off)
```

```
;;     idl-pad-keyword nil        ; Remove spaces for keyword assign
'='
;;         ;; If abbrev-mode is off, then case changes (the next 2
lines)
;;         ;; will not occur.
;;     idl-reserved-word-upcase t  ; Change reserved words to upper case
;;     idl-abbrev-change-case nil  ; Don't force case of expansions
;;     idl-hang-indent-regexp ": " ; Change from "- "
;;         idl-show-block nil         ; Turn off blinking to matching
begin
;;     idl-abbrev-move t          ; Allow abbrevs to move point
backwards
;;         case-fold-search nil       ; Make searches case sensitive
;;     )
;;
;;    ;; Run other functions here
;;        (font-lock-mode 1)          ; font-lock mode
;;    (idl-auto-fill-mode 0) ; Turn off auto filling
;;         ;; Pad with with 1 space (if -n is used then make the
;;         ;; padding a minimum of n spaces.)  The defaults use -1
;;         ;; instead of 1.
;;         (idl-action-and-binding "=" '(idl-expand-equal 1 1))
;;         (idl-action-and-binding "<" '(idl-surround 1 1))
;;         (idl-action-and-binding ">" '(idl-surround 1 1))
;;         (idl-action-and-binding "&" '(idl-surround 1 1))
;;         ;; Only pad after comma and with exactly 1 space
;;         (idl-action-and-binding "," '(idl-surround nil 1))
;;         ;; Set some personal bindings
;;         ;; (In this case, makes `,' have the normal self-insert
behavior.)
;;    (local-set-key "," 'self-insert-command)
;;         ;; Create a newline, indenting the original and new line.
;;         ;; A similar function that does _not_ reindent the original
;;         ;; line is on "\C-j" (The default for emacs programming
modes).
;;    (local-set-key "\n" 'idl-newline)
;;         ;; (local-set-key "\C-j" 'idl-newline) ; My preference.
;;    )))
;;
;; You can get a pop-up menu of the functions in an IDL file if you
;; have imenu.el.  Add the following to your .emacs to bind the menu
;; to shift mouse-button 3:
;;   (cond (window-system
;;         (define-key global-map [S-down-mouse-3] 'goto-index-pos))
;;
;; Note: In versions of imenu.el included with Emacs use "imenu" in
;; place of "goto-index-pos".
;;
```

```
;; You will automatically get an IDL menu of standard formatting
functions in
;; the main menu bar if you have easymenu.el in your Emacs loadpath.
;;
;; Highlighting of keywords, comments, and strings can be accomplished
;; with either font-lock or hilit19.  font-lock is preferred if have
;; it.  Font-lock mode is a minor mode, just as auto-fill mode is.
;; (But hilit19 is *not* a minor mode; it's hard to turn off, once
;; you've installed it!).  Use hilit19 if your system is too slow when
;; using font-lock.
;;
;; font-lock (Phil Williams, Ulrik Dickow, Simon Marshall):
;;
;; The default patterns are contained in 'idl-font-lock-keywords'.
;; You can customize these in your idl-mode-hook.
;;
;; To enable font-lock unconditionally in your idl-mode-hook place the
;; following line (see example hook above):
;;    (font-lock-mode 1)
;;
;;
;; hilit19: put this in your .emacs if hilit19.el is not loaded:
;;   (require 'hilit19)
;; The default patterns are contained in `idl-hilit-patterns'.
;; You can customize these in your idl-mode-hook.
;;
;; To enable hiliting for idl-mode, in your idl-mode-hook place the
;; following line (see above):
;;    (hilit-set-mode-patterns 'idl-mode idl-hilit-patterns nil t)
;;
;; Do NOT try to use both font-lock and hilit19 simultaneously.

;;; 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-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.")

(defvar idl-use-last-hang-indent nil
  "*If non-nil then use last match on line for `idl-indent-regexp'.")

(defvar idl-abbrev-start-char "\\"
"*A single character string used to start abbreviations in abbrev mode.
Possible characters to chose from: ~`\%
or even '?'.  '.' is not a good choice because it can make structure
field names act like abbrevs in certain circumstances.

Changes to this in idl-mode-hook will have no effect.  Instead a user
must set it directly using `setq' in the .emacs file before idl.el
is loaded.")


;;; font-lock mode - Additions by Phil Williams, Ulrik Dickow and
;;; Simon Marshall <simon@gnu.ai.mit.edu>..

(defvar idl-font-lock-keywords
  (if (or (save-match-data (string-match "Lucid" emacs-version))
   (not (boundp 'emacs-minor-version)) (< emacs-minor-version 30))
      ;;
      ;; For Emacs 19.29 and below, and Lucid/XEmacs.
      ;;
      ;;  At one time we had to include the capitalized versions,
      ;;  because allegedly abbrev-mode would remove the lock.  This
      ;;  problem neither occurred in Emacs nor XEmacs 19.13, but was
      ;;  seen in Lucid Emacs.
      ;;
      ;; reserved words
      (list
       (concat "\\<\\("
        "and\\|begin\\|c\\(ase\\|ommon\\)\\|do\\|"

" e\\(lse\\|nd\\(\\|case\\|else\\|for\\|if\\|rep\\|while\\)?\\ |q\\)\\| "
        "f\\(or\\|unction\\)\\|"
        "g\\([et]\\|oto\\)\\|if\\|l[et]\\|mod\\|n\\(e\\|ot\\)\\|"
```

```
        " o\\([fr]\\|n_ioerror\\)\\|pro\\|re\\(peat\\|turn\\)\\|then\\ | "
        "until\\|while\\|xor"
        "\\)\\>")
       ;;
       ;; Fontify function name.
       (cons (concat "\\<\\(pro\\|function\\)\\>[
 \t]+\\([a-zA-Z][a-zA-Z0-9$_]+\\(::\\)?\\([a-zA-Z][a-zA-Z0-9$ _]\\)?\\) ")
      '(2 font-lock-function-name-face nil t)))
    ;;
    ;; For Emacs 19.30 and up
    (list
     ;;
     ;; All keywords except below.
     (concat "\\<\\("
      "and\\|begin\\|case\\|do\\|e\\(lse\\|nd\\(\\|case\\|else\\|"
      "for\\|if\\|rep\\|while\\)\\|q\\)\\|for\\|"
      "g\\([et]\\|oto\\)\\|if\\|l[et]\\|mod\\|n\\(e\\|ot\\)\\|"
      "o\\([fr]\\|n_ioerror\\)\\|re\\(peat\\|turn\\)\\|then\\|"
      "until\\|while\\|xor"
      "\\)\\>")
     ;;
     ;; Function declarations.  Fontify keyword plus function name.
     '("\\<\\(function\\|pro\\)\\>[ \t]*\\(\\sw+\\(::\\sw+\\)?\\)?"
       (1 font-lock-keyword-face) (2 font-lock-function-name-face nil
t))
     ;;
     ;; Common declarations.  Fontify keyword, block name, plus variable
name(s).
  '("\\<\\(common\\)\\>[ \t]*\\(\\sw+\\)?,?"
       ;; Fontify the keyword.
       (1 font-lock-keyword-face)
       ;; Fontify the block name.
       (2 font-lock-reference-face nil t)
       ;; Fontify items as variable names.
        (font-lock-match-c++-style-declaration-item-and-skip-to-next nil
nil
        (1 font-lock-variable-name-face)))
     ))
  "Default expressions to highlight in IDL mode.")

(defvar idl-font-lock-defaults
  '(idl-font-lock-keywords nil t ((?$ . "w") (?_ . "w") (?. . "w"))
beginning-of-line))

;;; Variable for hilit.el to hilite IDL code

(defvar idl-hilit-patterns
       '((";.*" nil comment)
```

```
  ("\"[^\\\"\\\n]*\"" nil string)
  ("'[^'\\\n]*'" nil string)

(" \\S_\\(and\\|begin\\|case\\|common\\|do\\|else\\|end\\|endca
se\\|endelse\\|endfor\\|endif\\|endrep\\|endwhile\\|eq\\|for
\\|function\\|ge\\|goto\\|gt\\|if\\|le\\|lt\\|mod\\|ne\\|not
\\|of\\|on_ioerror\\|or\\|pro\\|repeat\\|return\\|then\\|unt il\\|while\\|xor\\)\\S_ "
nil keyword))
  "List of patterns and associated font faces for hilit.el.
See hilit19.el for documentation. ")

;;; Variables for abbrev and action behavior

(defvar idl-surround-by-blank nil
  "*If nil disables `idl-surround'.
If non-nil, `=',`<',`>',`&',`,' 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 and made into actions calling `idl-surround' with
`idl-action-and-binding'.
See help for `idl-action-and-binding' for examples.

Also see help for `idl-surround'.")

(defvar idl-pad-keyword t
  "*If non-nil then pad '=' for keywords like assignments.
Whenever `idl-surround' is non-nil then this affects how '=' is padded
for keywords.  If non-nil it is padded the same as for assignments.
If nil then spaces are removed.")

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

(defvar idl-do-actions nil
  "*If non-nil then performs actions when indenting.
The actions that can be performed are listed in
`idl-indent-action-table'.")

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

See `idl-check-abbrev'.")
```

```
(defvar idl-reserved-word-upcase nil
"*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'.
Has effect only if in abbrev-mode.")

(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.
Has effect only if in abbrev-mode.")

;;; Types of comments

(defvar idl-no-change-comment ";;;"
  "*The indentation of a comment that starts with this regular
expression will not be changed. Note that the indentation of a comment
at the beginning of a line is never changed.")

(defvar idl-begin-line-comment nil
  "*A comment anchored at the beginning of line.
A comment matching this regular expression will not have its
indentation changed.  If nil the default is \"^\;\", i.e., any line
beginning with a \"\;\".  Expressions for comments at the beginning of
the line should begin with \"^\".")

(defvar idl-code-comment ";;[^;]"
  "*A comment that starts with this regular expression on a line by
itself is indented as if it is a part of IDL code.  As a result if
the comment is not preceded by whitespace it is unchanged.")

;; Comments not matching any of the above will be indented as a
;; right-margin comment, i.e., to a minimum of `comment-column'.

;;; Action/Expand Tables.
;;
;; The average user may have difficulty modifying this directly.  It
;; can be modified/set in idl-mode-hook, but it is easier to use
;; idl-action-and-binding. See help for idl-action-and-binding for
;; examples of how to add an action.
;;
;; The action table is used by `idl-indent-line' whereas both the
;; action and expand tables are used by `idl-indent-and-action'.  In
;; general, the expand table is only used when a line is explicitly
;; indented.  Whereas, in addition to being used when the expand table
;; is used, the action table is used when a line is indirectly
```

```
;; indented via line splitting, auto-filling or a new line creation.
;;
;; Example actions:
;;
;;  Capitalize system vars
;;   (idl-action-and-binding idl-sysvar '(capitalize-word 1) t)
;;
;;  Capitalize procedure name
;;   (idl-action-and-binding "\\<\\(pro\\|function\\)\\>[ \t]*\\<"
;;          '(capitalize-word 1) t)
;;
;;  Capitalize common block name
;;   (idl-action-and-binding "\\<common\\>[ \t]+\\<"
;;         '(capitalize-word 1) t)
;;  Capitalize label
;;   (idl-action-and-binding (concat "^[ \t]*" idl-label)
;;                       '(capitalize-word -1) t)

(defvar idl-indent-action-table nil
  "*Associated array containing action lists of search string (car),
and function as a cdr. This table is used by `idl-indent-line'.
See documentation for `idl-do-action' for a complete description of
the action lists.

Additions to the table are made with `idl-action-and-binding' when a
binding is not requested.
See help on `idl-action-and-binding' for examples.")

(defvar idl-indent-expand-table nil
  "*Associated array containing action lists of search string (car),
and function as a cdr. The table is used by the
`idl-indent-and-action' function. See documentation for
`idl-do-action' for a complete description of the action lists.

Additions to the table are made with `idl-action-and-binding' when a
binding is requested.
See help on `idl-action-and-binding' for examples.")

;;; 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-timestamp-hook 'idl-default-insert-timestamp)

(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.")

;;; Miscellaneous variables

(defvar idl-fill-comment-line-only t
  "*If non-nil then auto fill will only operate on comment lines.")

(defvar idl-auto-fill-split-string t
"*If non-nil then auto fill will split a string with the IDL
string concatenation operator '+' if the point of splitting falls
inside a string.  If nil and a string is split then a terminal beep
and warning are issued.")

(defvar idl-split-line-string t
  "*If non-nil then `idl-split-line' will split a string with the IDL
string concatenation operator '+' if the point of splitting falls
inside a string.  If nil and a string is split then a terminal beep
and warning are issued.")

(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.")

;;;
;;; End customization variables section
;;;

;;; Non customization variables

(defconst 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.")

(defconst 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.")

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

(defconst 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.")

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

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

(defconst idl-identifier "[a-zA-Z][a-zA-Z0-9$_]+"
  "Regular expression matching an IDL identifier.")

(defconst idl-sysvar (concat "!" idl-identifier)
  "Regular expression matching IDL system variables.")

(defconst idl-variable (concat idl-identifier "\\|" idl-sysvar)
  "Regular expression matching IDL variable names.")

(defconst idl-label (concat idl-identifier ":")
```

```
  "Regular expression matching IDL labels.")

(defconst idl-statement-match
  (list
   ;; "endif else" is the the only possible "end" that can be
   ;; followed by a statement on the same line.
   '(endelse . ("end\\(\\|if\\)\\s +else" "end\\(\\|if\\)\\s +else"))
   ;; all other "end"s can not be followed by a statement.
   (cons 'end (list idl-end-block-reg nil))
   '(if . ("if\\>" "then"))
   '(for . ("for\\>" "do"))
   '(begin . ("begin\\>" nil))
   '(pdef . ("pro\\>\\|function\\>" nil))
   '(while . ("while\\>" "do"))
   '(repeat . ("repeat\\>" "repeat"))
   '(goto . ("goto\\>" nil))
   '(case . ("case\\>" nil))
   (cons 'call (list (concat idl-identifier "\\(\\s *$\\|\\s *,\\)")
nil))
   '(assign . ("[^=\n]*=" nil)))

   "Associated list of statement matching regular expresssions.
Each regular expression matches the start of an IDL statment.  The
first element of each association is a symbol giving the statement
type.  The associated value is a list.  The first element of this list
is a regular expression matching the start of an IDL statement for
identifying the statement type.  The second element of this list is a
regular expression for finding a substatement for the type.  The
substatement starts after the end of the found match modulo
whitespace.  If it is nil then the statement has no substatement.  The
list order matters since matching an assignment statement exactly is
not possible without parsing.  Thus assignment statement become just
the leftover unidentified statments containing and equal sign. "  )

(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.")

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

(defconst idl-mode-version "$Revision: 1.38 $")

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

;; If I take the time I can replace idl-keyword-abbrev with
;; idl-code-abbrev and remove the quoted abbrev check from
;; idl-check-abbrev.  Then, e.g, (idl-keyword-abbrev 0 t) becomes
;; (idl-code-abbrev idl-check-abbrev 0 t).  In fact I should change
;; the name of idl-check-abbrev to something like idl-modify-abbrev.

(defmacro idl-code-abbrev (&rest args)
  "Creates a function for abbrev hooks that ensures abbrevs are not
quoted.
Specifically, if the abbrev is in a comment or string it is unexpanded.
Otherwise ARGS forms a list that is evaluated."
  (` (quote (lambda ()
      (if (idl-quoted) (progn (unexpand-abbrev) nil)
  (, (append args)))))))

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

(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 ?_   "_"  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 ?\n  ">"  idl-mode-syntax-table)
  (modify-syntax-entry ?\f  ">"  idl-mode-syntax-table))

(defvar idl-find-symbol-syntax-table nil
  "Syntax table that treats symbol characters as word characters.")

(if idl-find-symbol-syntax-table
    ()
  (setq idl-find-symbol-syntax-table
 (copy-syntax-table idl-mode-syntax-table))
  (modify-syntax-entry ?$   "w"  idl-find-symbol-syntax-table)
  (modify-syntax-entry ?_   "w"  idl-find-symbol-syntax-table))


(defun idl-action-and-binding (key cmd &optional select)
  "KEY and CMD are made into a key binding and an indent action.
KEY is a string - same as for the `define-key' function.  CMD is a
function of no arguments or a list to be evalauated.  CMD is bound to
KEY in `idl-mode-map' by defining an anonymous function calling
`self-insert-command' followed by CMD.  If KEY contains more than one
character a binding will only be set if SELECT is \'both.

\(KEY . CMD\) is also placed in the `idl-indent-expand-table',
replacing any previous value for KEY.  If a binding is not set then it
will instead be placed in `idl-indent-action-table'.

If the optional argument SELECT is nil then an action and binding are
created.  If SELECT is \'noaction, then a binding is always set and no
action is created.  If SELECT is \'both then an action and binding
will both be created even if KEY contains more than one character.
Otherwise, if SELECT is non-nil then only an action is created.

Some examples:
\; No spaces before and 1 after a comma
```

```
\(idl-action-and-binding \",\"  '\(idl-surround 0 1\)\)
\; A minimum of 1 space before and after `=' \(see `idl-expand-equal'\).
\(idl-action-and-binding \"=\"  '\(idl-expand-equal -1 -1\)\)
\; Capitalize system variables - action only
\(idl-action-and-binding idl-sysvar '\(capitalize-word 1\) t\)"
  (if (not (equal select 'noaction))
      ;; Add action
      (let* ((table (if select 'idl-indent-action-table
        'idl-indent-expand-table))
      (cell (assoc key (eval table))))
 (if cell
     ;; Replace action command
     (setcdr cell cmd)
   ;; New action
   (set table (append (eval table) (list (cons key cmd)))))))
  ;; Make key binding for action
  (if (or (and (null select) (= (length key) 1))
   (equal select 'noaction)
   (equal select 'both))
      (define-key idl-mode-map key
 (append '(lambda ()
       (interactive)
       (self-insert-command 1))
  (list (if (listp cmd)
     cmd
   (list cmd)))))))

(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 "\M-\t"    'idl-hard-tab)
  (define-key idl-mode-map "\C-c\;" '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-h"  'idl-doc-header)
  (define-key idl-mode-map "\C-c\C-m"  'idl-doc-modification)
  (define-key idl-mode-map "\C-c\C-c"  'idl-case)
  (define-key idl-mode-map "\C-c\C-d" 'idl-debug-map)
  (define-key idl-mode-map "\C-c\C-f" 'idl-for)
  ;;  (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))

;; Set action and key bindings.
;; See description of the function `idl-action-and-binding'.
;; Automatically add spaces for the following characters
(idl-action-and-binding "&"  '(idl-surround -1 -1))
(idl-action-and-binding "<"  '(idl-surround -1 -1))
(idl-action-and-binding ">"  '(idl-surround -1 -1))
(idl-action-and-binding ","  '(idl-surround 0 -1))
;; Automatically add spaces to equal sign if not keyword
(idl-action-and-binding "="  '(idl-expand-equal -1 -1))

;;;
;;; Abbrev Section
;;;
;;; When expanding abbrevs and the abbrev hook moves backward, an extra
;;; space is inserted (this is the space typed by the user to expanded
;;; the abbrev).
;;;

(condition-case nil
    (modify-syntax-entry (string-to-char idl-abbrev-start-char) "w"
idl-mode-syntax-table)
  (error nil))

(defvar idl-mode-abbrev-table nil)
(if idl-mode-abbrev-table
    ()
  (define-abbrev-table 'idl-mode-abbrev-table ())
  (let ((abbrevs-changed nil))
    ;;
    ;; Templates
    ;;
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
```

```
"c")   "" (idl-code-abbrev
idl-case))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"f")   "" (idl-code-abbrev
idl-for))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"fu")  "" (idl-code-abbrev
idl-function))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"pr")  "" (idl-code-abbrev
idl-procedure))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"r")   "" (idl-code-abbrev
idl-repeat))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"w")   "" (idl-code-abbrev
idl-while))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"i")   "" (idl-code-abbrev
idl-if))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"elif")   "" (idl-code-abbrev
idl-elif))
    ;;
    ;; Keywords, system functions, conversion routines
    ;;
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"b")   "begin"
(idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"co")  "common"
(idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cb")  "byte()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cx")  "fix()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cl")  "long()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cf")  "float()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cs")  "string()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
```

```
"cc")  "complex()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"cd")  "double()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"e")   "else"
(idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ec")  "endcase"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ee")  "endelse"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ef")  "endfor"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ei")  "endif else if"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"el")  "endif else"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"en")  "endif"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"er")  "endrep"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ew")  "endwhile"
'idl-show-begin)
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"g")   "goto,"
(idl-keyword-abbrev 0 t))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"h")   "help,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"k")   "keyword_set()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"n")   "n_elements()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"on")  "on_error,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
```

```
"oi")  "on_ioerror,"
(idl-keyword-abbrev 0 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ow")  "openw,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"or")  "openr,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ou")  "openu,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"p")  "print,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"pt")  "plot,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"re")  "read,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"rf")  "readf,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ru")  "readu,"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"rt")  "return"
(idl-keyword-abbrev 0))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sc")  "strcompress()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sn")  "strlen()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sl")  "strlowcase()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"su")  "strupcase()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sm")  "strmid()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sp")  "strpos()"
(idl-keyword-abbrev 1))
   (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
```

```
"st")  "strput()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"sr")  "strtrim()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"t")   "then"
(idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"u")   "until"
(idl-keyword-abbrev 0 t))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"wu")  "writeu,"
(idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"ine") "if n_elements() eq 0 then"
(idl-keyword-abbrev 11))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"inn") "if n_elements() ne 0 then"
(idl-keyword-abbrev 11))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"np")  "n_params()"
(idl-keyword-abbrev 0))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"s")   "size()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"wi")  "widget_info()"
(idl-keyword-abbrev 1))
    (define-abbrev idl-mode-abbrev-table  (concat idl-abbrev-start-char
"wc")  "widget_control,"
(idl-keyword-abbrev 0))

    ;; 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-check)
   (define-abbrev idl-mode-abbrev-table "endcase"  "endcase"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endelse"  "endelse"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endfor"   "endfor"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endif"    "endif"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endrep"   "endrep"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endwhi"   "endwhi"
'idl-show-begin-check)
   (define-abbrev idl-mode-abbrev-table "endwhile" "endwhile"
'idl-show-begin-check)
   (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:

  Like other Emacs programming modes, \\C-j inserts a newline and
  indents.  TAB is used for explicit indentation of the current line.

  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
    `idl-continuation-indent' spaces after rightmost unmatched 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 and at the beginning of the line.

    Actions can be performed when a line is indented, e.g., surrounding
    `=' and `>' with spaces.  See help for `idl-indent-action-table'.

  Comment Indentation:

    Controlled by customizable varibles that match the start of a
comment.
    Indentation for comments beginning with \(default settings\):

    1\) \; in first column - unchanged (see `idl-begin-line-comment').
    2\) `idl-no-change-comment' \(\";;;\"\) - indentation is not
```

changed.
    3\) `idl-code-comment' \(\";;[^;]\"\ i.e. exactly two `;' \) -
indented as IDL code.
    4\) None of the above \(i.e. single `\;' not in first column) -
indented
  to a minimum of `comment-column' (called a right-margin comment).

  Indentation of text inside a comment paragraph:

    A line whose first non-whitespace character is `\;' is called
    a `comment line'.

    This mode handles comment paragraphs to a limited degree.
    Specifically, comment paragraphs only have meaning for auto-fill
    and the idl-fill-paragraph command \\[idl-fill-paragraph].  A
    comment line is considered blank if there is only whitespace
    besides the comment delimeters.  A comment paragraph consists of
    consecutive nonblank comment lines containing the same comment
    leader (the whitespace at the beginning of the line plus comment
    delimiters). Thus, blank comment lines or a change in the comment
    leader 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.

    To fill the current comment paragraph use idl-fill-paragraph,
\\[idl-fill-paragraph].

\; 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.

  Indentation in auto-fill-mode:

When in auto-fill mode code lines are continued and indented appropriately. Comments within a comment paragraph are wrapped and indented. The comment text is indented as explained above. To toggle on/off the auto-fill mode use `idl-auto-fill-mode', \\[idl-auto-fill-mode], rather than the normal `auto-fill-mode' function.

Variables controlling indentation style and other 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-surround-by-blank'
Automatically surround '=','<','>' 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 make hanging indents. (default t)
`idl-indent-action-table'
An associated list of strings to match and commands to perform when indenting a line. Enabled with `idl-do-actions'. To make additions use `idl-action-and-binding' (which can also be used to make key bindings).

Other features:

Use `M-x list-abbrevs' to display a list of built-in abbreviation expansions when abbrev-mode is turned on.

The case of reserved words and some 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', C-c C-m or \\[idl-doc-modification].

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

Thus you can se any desired customizations by adding a function to
this hook.

To convert pre-existing IDL code to your formatting style, mark the
entire buffer with `mark-whole-buffer' (\\[mark-whole-buffer]) then
execute `idl-expand-region-abbrevs'.  Then mark the entire buffer
again
followed by `indent-region' (\\[indent-region]).

Key bindings:

Many templates for 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 key map and work analogously to
the list commands.  Use \\[idl-split-line] to continue or split a
code line or a comment.

If some of the key bindings below show with ??, use \\[describe-key]
followed by the key sequence to see what the key sequence does.

\\{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-and-action)
  (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)
  (if (featurep 'easymenu)
      (easy-menu-add idl-menu idl-mode-map))
  (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 'comment-multi-line)
  (setq comment-multi-line nil)
  (make-local-variable 'paragraph-separate)
  (setq paragraph-separate "[ \t\f]*$\\|[ \t]*;+[ \t]*$")
  (make-local-variable 'paragraph-start)
  (setq paragraph-start "[ \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)
  ;;
  ;; Set tag table list to use IDLTAGS
  (setq tag-table-alist (append '(("\\.pro$" . "IDLTAGS"))))
  ;; font-lock additions - originally Phil Williams, then Ulrik Dickow
  ;;
  (make-local-variable 'font-lock-defaults)
  (setq font-lock-defaults idl-font-lock-defaults)
;  (make-local-variable 'font-lock-keywords)
;  (setq font-lock-keywords idl-font-lock-keywords)
;  (make-local-variable 'font-lock-keywords-case-fold-search) ; Emacs
;  (setq font-lock-keywords-case-fold-search t) ; Emacs
;  (put major-mode 'font-lock-keywords-case-fold-search t) ; XEmacs
(/Lucid?)
  (put major-mode 'font-lock-defaults idl-font-lock-defaults) ; XEmacs
(/Lucid?)
  (if (or (featurep 'imenu) (load "imenu" t))
      (progn
 (setq imenu-create-index-function
       (function imenu-default-create-index-function))
 ;; Handle name changes for imenu.el
 (if (fboundp 'goto-index-pos)
     (setq
      extract-index-name-function (function idl-unit-name)
      prev-index-position-function (function idl-prev-index-position))
   ;; Names for the version included with the Emacs release
   (setq
    imenu-extract-index-name-function (function idl-unit-name)
    imenu-prev-index-position-function
    (function idl-prev-index-position)))))

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

;;
```

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

(defun idl-push-mark (&rest rest)
  "Push mark for compatibility with Emacs 18/19."
  (if (fboundp 'iconify-frame)
      (apply 'push-mark rest)
    (push-mark)))

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

;;; This stuff is experimental.

(defvar idl-command-hook nil
  "If non-nil, a list that can be evaluated using `eval'.
It is evaluated in the lisp function `idl-command-hook' which is
placed in `post-command-hook'.")

(defun idl-command-hook ()
  "Command run after every command"
  (if (listp idl-command-hook)
      (eval idl-command-hook))
  (setq idl-command-hook nil))

(if (boundp 'post-command-hook)
    (add-hook 'post-command-hook 'idl-command-hook))

;;; End experiment

;; It would be better to use expand.el for better abbrev handling and
;; versatility.

(defun idl-check-abbrev (arg &optional reserved)
  "Reverses abbrev expansion if in comment or string.
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.
Otherwise, the abbrev will be capitalized if `idl-abbrev-change-case'
is non-nil, unless its value is \`down in which case the abbrev will be
made into all lowercase.
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))
      (cond
       ((equal idl-abbrev-change-case 'down)
 (downcase-region last-abbrev-location (point)))
       (idl-abbrev-change-case
 (upcase-region last-abbrev-location (point)))))
    (if (and idl-abbrev-move (> arg 0))
 (if (boundp 'post-command-hook)
     (setq idl-command-hook (list 'backward-char (1+ arg)))
   (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)))
 (data (match-data))
 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)))
    (store-match-data data)
    (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-check ()
  "Ensure that the previous word was a token before `idl-show-begin'.
An END token must be preceded by whitespace."
  (if
      (save-excursion
 (backward-word 1)
 (backward-char 1)
 (looking-at "[ \t\n\f]"))
      (idl-show-begin)))

(defun idl-show-begin ()
  "Finds the start of current block and blinks to it for a second."
  ;; All end statements are reserved words
  (if (idl-check-abbrev 0 t)
      (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. See description of `idl-make-space'.

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
 (backward-char 1)
 (save-restriction
   (let ((here (point)))
     (skip-chars-backward " \t")
```

```
    (if (bolp)
  ;; avoid clobbering indent
  (progn
    (move-to-column (idl-calculate-indent))
    (if (<= (point) here)
        (narrow-to-region (point) here))
    (goto-char here)))
     (idl-make-space before))
   (skip-chars-forward " \t"))
 (forward-char 1)
 (idl-make-space after)
 ;; Check to see if the line should auto wrap
 (if (and (equal (char-after (1- (point))) ? )
   (> (current-column) fill-column))
     (funcall auto-fill-function)))))

(defun idl-make-space (n)
  "Make space at point.
The space affected is all the spaces and tabs around point.
If n is non-nil then point is left abs\(n\) spaces from the beginning of
the contiguous space.
The amount of space at point is determined by N.
If the value of N is:
nil   - do nothing.
c > 0 - exactly c spaces.
c < 0 - a minimum of -c spaces, i.e., do not change if there are
        already -c spaces.
0     - no spaces."
  (if (integerp n)
      (let
   ((start-col (progn (skip-chars-backward " \t") (current-column)))
    (left (point))
    (end-col (progn (skip-chars-forward " \t") (current-column))))
 (delete-horizontal-space)
 (cond
  ((> n 0)
   (idl-indent-to (+ start-col n))
   (goto-char (+ left n)))
  ((< n 0)
   (idl-indent-to end-col (- n))
   (goto-char (- left n)))
  ;; n = 0, done
  ))))

(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)
  ;;
  ;; The current line is being split, the cursor should be at the
  ;; beginning of the new line skipping the leading indentation.
  ;;
  ;; The reason we insert the new line before indenting is that the
  ;; indenting could be confused by keywords (e.g. END) on the line
  ;; after the split point.  This prevents us from just using
  ;; `indent-for-tab-command' followed by `newline-and-indent'.
  ;;
  (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 (or (looking-at idl-no-change-comment)
   (if idl-begin-line-comment
       (looking-at idl-begin-line-comment)
       (looking-at "^\;")))
      (current-column)
    (if (looking-at idl-code-comment)
 (if (save-excursion (skip-chars-backward " \t") (bolp))
     ;; On line by itself, indent as code
     (let ((tem (idl-calculate-indent)))
       (if (listp tem) (car tem) tem))
   ;; after code - do not change
   (current-column))
      (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 line
```

```
comment
then the new line will contain a comment with the same indentation.
Splits strings with the IDL operator `+' if `idl-split-line-string' is
non-nil."
  (interactive)
  (let (beg)
    (if (not (idl-in-comment))
 ;; For code line add continuation.
 ;; Check if splitting a string.
 (progn
   (if (setq beg (idl-in-quote))
       (if idl-split-line-string
    ;; Split the string.
    (progn (insert (setq beg (char-after beg)) " + "
     idl-continuation-char beg)
    (backward-char 1))
  ;; Do not split the string.
  (beep)
  (message "Warning: continuation inside string!!")
  (insert " " idl-continuation-char))
     ;; Not splitting a string.
     (insert " " idl-continuation-char))
   (newline-and-indent))
      (indent-new-comment-line))
    ;; Indent previous line
    (setq beg (- (point-max) (point)))
    (forward-line -1)
    (idl-indent-line)
    (goto-char (- (point-max) beg))
    ;; Reindent new line
    (idl-indent-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-statement ()
  "Mark current IDL statement."
  (interactive)
  (idl-end-of-statement)
  (let ((end (point)))
```

```
    (idl-beginning-of-statement)
    (idl-push-mark end nil t)))

(defun idl-mark-block ()
  "Mark containing block."
  (interactive)
  (idl-end-of-statement)
  (idl-backward-up-block -1)
  (idl-end-of-statement)
  (let ((end (point)))
    (idl-backward-block)
    (idl-beginning-of-statement)
    (idl-push-mark end nil t)))


(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)
  (let ((beg (point)))
    (idl-forward-block)
    (idl-push-mark beg nil t))
  (exchange-point-and-mark))

(defun idl-backward-up-block (&optional arg)
  "Move to beginning of enclosing block if prefix 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)
       (idl-push-mark beg nil t)
   (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 line comments
      (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]*$")
       (looking-at (concat "[ \t]*" idl-label "[ \t]*$"))
       (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 line comments
    (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 (concat "[ \t]*" idl-label "[ \t]*$"))
        (looking-at "^@")))))
    last-statement))

(defun idl-skip-label ()
  "Skip label or case statement element.
Returns position after label.
If there is no label point is not moved and nil is returned."
  ;; Just look for the first non quoted colon and check to see if it
  ;; is inside a sexp.  If is not in a sexp it must be part of a label
  ;; or case statement element.
  (let ((start (point))
 (end (idl-find-key ":" 1 'nomark
      (save-excursion
        (idl-end-of-statement) (point)))))
    (if (and end
      (= (nth 0 (parse-partial-sexp start end)) 0))
 (progn
   (forward-char)
   (point))
      (goto-char start)
      nil)))

(defun idl-start-of-substatement (&optional pre)
  "Move to start of next IDL substatement after point.
Uses the type of the current IDL statement to determine if the next
statement is on a new line or is a subpart of the current statement.
Returns point at start of substatement modulo whitespace.
If optional argument is non-nil move to beginning of current
substatement. "
  (let ((orig (point))
 (eos (idl-end-of-statement))
 (ifnest 0)
 st nst last)
    (idl-beginning-of-statement)
    (idl-skip-label)
    (setq last (point))
    ;; Continue looking for substatements until we are past orig
    (while (and (<= (point) orig) (not (eobp)))
      (setq last (point))
      (setq nst (nth 1 (cdr (setq st (car (idl-statement-type))))))
      (if (equal (car st) 'if) (setq ifnest (1+ ifnest)))
      (cond ((and nst
    (idl-find-key nst 1 'nomark eos))
      (goto-char (match-end 0)))
     ((and (> ifnest 0) (idl-find-key "\\<else\\>" 1 'nomark eos))
      (setq ifnest (1- ifnest))
      (goto-char (match-end 0)))
```

```
     (t (setq ifnest 0)
        (idl-next-statement))))
    (if pre (goto-char last))
    (point)))

(defun idl-statement-type ()
  "Return the type of the current IDL statement.
Uses `idl-statement-match' to return a cons of \(type . point\) with
point the ending position where the type was determined. Type is the
association from `idl-statment-match', i.e. the cons cell from the
list not just the type symbol. Returns nil if not an identifiable
statement."
  (save-excursion
    ;; Skip whitespace within a statement which is spaces, tabs,
continuations
    (while (looking-at "[ \t]*\\<\\$")
      (forward-line 1))
    (skip-chars-forward " \t")
    (let ((st idl-statement-match)
   (case-fold-search t))
      (while (and (not (looking-at (nth 0 (cdr (car st)))))
    (setq st (cdr st))))
      (if st
   (append st (match-end 0))))))

(defun idl-expand-equal (&optional before after)
  "Pad '=' with spaces.
Two cases: Assignment statement, and keyword assignment.
The case is determined using `idl-start-of-substatement' and
`idl-statement-type'.
The equal sign will be surrounded by BEFORE and AFTER blanks.
If `idl-pad-keyword' is non-nil then keyword
assignment is treated just like assignment statements.  Otherwise,
spaces are removed for keyword assignment.
Limits in for loops are treated as keyword assignment.
See `idl-surround'. "
  ;; Even though idl-surround checks `idl-surround-by-blank' this
  ;; check saves the time of finding the statement type.
  (if idl-surround-by-blank
      (let ((st (save-excursion
    (idl-start-of-substatement t)
    (idl-statement-type))))
 (if (or
      (and (equal (car (car st)) 'assign)
    (equal (cdr st) (point)))
      idl-pad-keyword)
     ;; An assignment statement
     (idl-surround before after)
```

```
   (idl-surround 0 0)))))

(defun idl-indent-and-action ()
  "Call `idl-indent-line' and do expand actions."
  (interactive)
  (idl-indent-line t)
  )

(defun idl-indent-line (&optional expand)
  "Indents current IDL line as code or as a comment.
The actions in `idl-indent-action-table' are performed.
If the optional argument EXPAND is non-nil then the actions in
`idl-indent-expand-table' are performed."
  (interactive)
  ;; Move point out of left margin.
  (if (save-excursion
 (skip-chars-backward " \t")
 (bolp))
      (skip-chars-forward " \t"))
  (let ((mloc (point-marker)))
    (save-excursion
      (beginning-of-line)
      (if (looking-at idl-comment-line-start-skip)
   ;; Indentation for a line comment
   (progn
     (skip-chars-forward " \t")
     (idl-indent-left-margin (idl-comment-hook)))
 ;;
 ;; Code Line
 ;;
 ;; Before indenting, run action routines.
 ;;
 (if (and expand idl-do-actions)
     (mapcar 'idl-do-action idl-indent-expand-table))
 ;;
 (if idl-do-actions
     (mapcar 'idl-do-action idl-indent-action-table))
 ;;
 ;; No longer expand abbrevs on the line.  The user can do this
 ;; manually using expand-region-abbrevs.
 ;;
 ;; Indent for code line
 ;;
 (beginning-of-line)
 (if (or
      ;; a label line
      (looking-at (concat "^" idl-label "[ \t]*$"))
      ;; a batch command
```

```
      (looking-at "^[ \t]*@"))
     ;; leave flush left
     nil
   ;; indent the line
   (idl-indent-left-margin (idl-calculate-indent)))
 ;; Adjust parallel comment
 (end-of-line)
 (if (idl-in-comment)
     (indent-for-comment))))
   (goto-char mloc)
   ;; Get rid of marker
   (set-marker mloc nil)
   ))

(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 point
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 &optional min)
  "Indent from point with spaces until column COL.
Inserts space before markers at point."
  (if (not min) (setq min 0))
  (insert-before-markers
   (make-string (max min (- col (current-column))) ? )))

(defun idl-indent-left-margin (col)
  "Indent the current line to column COL.
Indents such that first non-whitespace character is at column COL
Inserts spaces before markers at point."
  (save-excursion
    (beginning-of-line)
    (delete-horizontal-space)
    (idl-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 t)
     (+ (idl-current-statement-indent) idl-main-block-indent))
    ;; Begin block
    ((idl-look-at idl-begin-block-reg t)
     (+ (idl-current-statement-indent) idl-block-indent))
    ((idl-look-at idl-end-block-reg t)
     (- (idl-current-statement-indent) idl-end-offset idl-block-indent))
    ((idl-current-statement-indent))))))
   ;; adjust the indentation based on the current statement
   (cond
    ;; End block
    ((idl-look-at idl-end-block-reg t)
     (+ 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
    (case-fold-search t)
    (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-sexp 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
 (old-syntax-table (syntax-table))
 (case-fold-search t))
    (set-syntax-table idl-find-symbol-syntax-table)
    (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))))))
    (set-syntax-table old-syntax-table)
    (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))
  (case-fold-search t)
  (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-statement-indent ()
  "Return indentation of the current statement.
If in a statement, moves to beginning of statement before finding
indent."
```

```
  (idl-beginning-of-statement)
  (idl-current-indent))

(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))
 (old-syntax-table (syntax-table))
 (case-fold-search t)
 eos
 found)
    (set-syntax-table idl-find-symbol-syntax-table)
    (setq eos
   (if cont
       (save-excursion (idl-end-of-statement) (point))
     (save-excursion (end-of-line) (point))))
    (if beg (idl-beginning-of-statement))
    (while (and (setq found (re-search-forward regexp eos t))
  (idl-quoted)))
    (set-syntax-table old-syntax-table)
    (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
line comments. 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 line comment
  (if (save-excursion
 (beginning-of-line)
 (skip-chars-forward " \t")
 (looking-at comment-start))
      (let
   ((indent 999)
    pre here diff fill-prefix-reg bcl first-indent
    comment-leader hang start end)
 ;; Change tabs to spaces in the surrounding paragraph.
 ;; The surrounding paragraph will be the largest containing block of
 ;; contiguous line comments. 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)
 (setq bcl (point))
 (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))
 ;; the comment leader is the indentation plus exactly the
 ;; number of consecutive ";".
```

```
(setq fill-prefix-reg
      (concat
       (setq fill-prefix
      (regexp-quote
       (buffer-substring (save-excursion
      (beginning-of-line) (point))
    (point))))
       "[^;]"))
(setq comment-leader fill-prefix)
;; Mark the beginning and end of the paragraph
(goto-char bcl)
(while (and (looking-at fill-prefix-reg)
     (not (looking-at paragraph-separate))
     (not (bobp)))
  (forward-line -1))
;; Move to first line of paragraph
(if (/= (point) bcl)
    (forward-line 1))
(setq start (point))
(goto-char bcl)
(while (and (looking-at fill-prefix-reg)
     (not (looking-at paragraph-separate))
     (not (eobp)))
  (forward-line 1))
(beginning-of-line)
(if (or (not (looking-at fill-prefix-reg))
 (looking-at paragraph-separate))
    (forward-line -1))
(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)
          ? )))
;; first-line indent
(setq first-indent
```

```
       (max
        (progn
   (re-search-forward
    comment-start-skip
    (save-excursion (end-of-line) (point))
    t)
   (current-column))
       indent))

 ;; try to keep point at its original place
 (goto-char here)
; ;; fill the paragraph
; ;; This version of fill-region-as-paragraph is only
; ;; available in GNU emacs 19.31 or later (and not in
; ;; Xemacs 19.14)
; (save-excursion
;   (fill-region-as-paragraph
;    start
;    (- (point-max) end)
;    (current-justification)
;    nil
;    (+ start indent -1)))
 ;; In place of the more modern fill-region-as-paragraph, a hack
 ;; to keep whitespace untouched on the first line within the
 ;; indent length and to preserve any indent on the first line
 ;; (first indent).
 (save-excursion
   (setq diff
  (buffer-substring start (+ start first-indent -1)))
   (subst-char-in-region start (+ start first-indent -1) ?  ?~ nil)
   (fill-region-as-paragraph
    start
    (- (point-max) end)
    (current-justification)
    nil)
   (delete-region start (+ start first-indent -1))
   (goto-char start)
   (insert diff))
 ;; 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 the first match
with the `idl-hang-indent-regexp'.  If `idl-use-last-hang-indent' is
non-nil then use last occurrence matching `idl-hang-indent-regexp' on
```

```
the line.
If not found returns nil."
  (if idl-use-last-hang-indent
      (save-excursion
 (end-of-line)
 (if (re-search-backward
      idl-hang-indent-regexp
      (save-excursion (beginning-of-line) (point))
      t)
     (+ (current-column) (length idl-hang-indent-regexp))))
    (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.
Only fills comment lines if `idl-fill-comment-line-only' is non-nil.
Places a continuation character at the end of the line
if not in a comment.  Splits strings with IDL concantenation operator
`+' if `idl-auto-fill-split-string is non-nil."
    (if (<= (current-column) fill-column)
 nil    ; do not to fill
      (if (or (not idl-fill-comment-line-only)
       (save-excursion
  ;; Check for comment line
  (beginning-of-line)
  (looking-at idl-comment-line-start-skip)))
   (let (beg)
     (idl-indent-line)
     ;; Prevent actions for do-auto-fill which calls
indent-line-function.
     (let (idl-do-actions
    (paragraph-start ".")
    (paragraph-separate "."))
       (do-auto-fill))
     (save-excursion
       (end-of-line 0)
       ;; Indent the split line
       (idl-indent-line)
       )
     (if (save-excursion
    (beginning-of-line)
    (looking-at idl-comment-line-start-skip))
  ;; A continued line comment
```

```
;; We treat continued line comments as part of a comment
;; paragraph. So we check for a hanging indent.
(if idl-hanging-indent
    (let ((here (- (point-max) (point)))
   (indent
    (save-excursion
      (forward-line -1)
      (idl-calc-hanging-indent))))
      (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)
     (delete-horizontal-space)
     (idl-indent-to indent)
     (goto-char (- (point-max) here)))
 )))
     ;; Split code or comment?
     (if (save-excursion
    (end-of-line 0)
    (idl-in-comment))
  ;; Splitting a non-line comment.
  ;; Insert the comment delimiter from split line
  (progn
    (save-excursion
      (beginning-of-line)
      (skip-chars-forward " \t")
      ;; Insert blank to keep off beginning of line
      (insert " "
       (save-excursion
  (forward-line -1)
  (buffer-substring (idl-goto-comment)
     (progn (skip-chars-forward "; ")
     (point))))))
    (idl-indent-line))
;; Split code line - add continuation character
(save-excursion
  (end-of-line 0)
  ;; Check to see if we split a string
  (if (and (setq beg (idl-in-quote))
    idl-auto-fill-split-string)
      ;; Split the string and concatenate.
      ;; The first extra space is for the space
      ;; the line was split. That space was removed.
      (insert " " (char-after beg) " +"))
```

```
    (insert " $"))
  (if beg
      (if idl-auto-fill-split-string
   ;; Make the second part of continued string
   (save-excursion
     (beginning-of-line)
     (skip-chars-forward " \t")
     (insert (char-after beg)))
       ;; Warning
       (beep)
       (message "Warning: continuation inside a string.")))
  ;; 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))
      )))))

(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-default-insert-timestamp ()
  "Default timestamp insertion function"
  (insert (current-time-string))
  (insert ", " (user-full-name))
  (insert " <" (user-login-name) "@" (system-name) ">")
  ;; Remove extra spaces from line
  (idl-fill-paragraph)
  ;; Insert a blank line comment to separate from the date entry -
  ;; will keep the entry from flowing onto date line if re-filled.
  (insert "\n;\n;\t\t"))

(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")
```

```
  (run-hooks 'idl-timestamp-hook))

;;; 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))
    (data (match-data))
    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)))))
   (store-match-data data)
     ;; return string beginning position or nil
     (if (> start bq) bq))))

;; Statement templates

;; Replace these with a general template function, something like
;; expand.el (I think there was also something with a name similar to
;; dmacro.el)

(defun idl-template (s1 s2 &optional prompt noindent)
  "Build a template with optional prompt expression.

Opens a line if point is not followed by a newline modulo intervening
whitespace.  S1 and S2 are strings.  S1 is inserted at point followed
by S2.  Point is inserted between S1 and S2.  If optional argument
PROMPT is a string then it is displayed as a message in the
minibuffer.  The PROMPT serves as a reminder to the user of an
expression to enter.
```

```
The lines containing S1 and S2 are reindented using `indent-region'
unless the optional second argument NOINDENT is non-nil."
  (let ((beg (save-excursion (beginning-of-line) (point)))
 end)
    (if (not (looking-at "\\s-*\n"))
 (open-line 1))
    (insert s1)
    (save-excursion
      (insert s2)
      (setq end (point)))
    (if (not noindent)
 (indent-region beg end nil))
    (if (stringp prompt)
 (message prompt))))

(defun idl-elif ()
  "Build skeleton IDL if-else block."
  (interactive)
  (idl-template "if" " then begin\n\nendif else begin\n\nendelse"
"Condition expression"))

(defun idl-case ()
  "Build skeleton IDL case statement."
  (interactive)
  (idl-template "case" " of\n\nendcase" "Selector expression"))

(defun idl-for ()
  "Build skeleton for loop statment."
  (interactive)
  (idl-template "for" " do begin\n\nendfor" "Loop expression"))

(defun idl-if ()
  "Build skeleton for loop statment."
  (interactive)
  (idl-template "if" " then begin\n\nendif" "Scalar logical
expression"))

(defun idl-procedure ()
  (interactive)
  (idl-template "pro" "\n\nreturn\nend" "Procedure name"))

(defun idl-function ()
  (interactive)
  (idl-template "function" "\n\nreturn\nend" "Function name"))

(defun idl-repeat ()
  (interactive)
```

```
  (idl-template "repeat begin\n\nendrep until" "" "Exit condition"))

(defun idl-while ()
  (interactive)
  (idl-template "while" " do begin\n\nendwhile" "Entry condition"))

(defun idl-split-string (string)
  (setq start 0)
  (setq last (length string))
  (setq lst ())
  (while (setq end (string-match "[ \t]+" string start))
    (setq lst (append lst (list (substring string start end))))
    (setq start (match-end 0)))
    (setq lst (append lst (list (substring string start last)))))

(defun idl-replace-string (string replace_string replace_with)
  (setq start 0)
  (setq last (length string))
  (setq ret_string "")
  (while (setq end (string-match replace_string string start))
    (setq ret_string
   (concat ret_string (substring string start end) replace_with))
    (setq start (match-end 0)))
    (setq ret_string (concat ret_string (substring string start last))))

(defun idl-make-tags ()
  "Creates the IDL tags file IDLTAGS in the current directory from
the list of directories specified in the minibuffer. Directories may be
for example: . /usr/local/rsi/idl/lib. All the subdirectories of the
specified top directories are searched if the directory name is prefixed
by @. Specify @ directories with care, it may take a long, long time if
you specify /."
  (interactive)
  ;;
  ;; Read list of directories
  (setq directory (read-string "Tag Directories: " "."))
  (setq directories (idl-split-string directory))
  ;;
  ;; Set etags command, vars
  (setq cmd "etags --output=IDLTAGS --language=none --regex='/[
\\t]*[pP][Rr][Oo][ \\t]+\\([^ \\t,]+\\)/' --regex='/[
\\t]*[Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn][ \\t]+\\([^ \\t,]+\\)/' ")
  (setq append " ")
  (setq status 0)
  ;;
  ;; For each directory
  (setq numdirs 0)
  (setq dir (nth numdirs directories))
```

```
 (while (and dir)
   ;;
   ;; Find the subdirectories
   (if (string-match "^[@]\\(.+\\)$" dir)
(setq getsubdirs t) (setq getsubdirs nil))
   (if (and getsubdirs) (setq dir (substring dir 1 (length dir))))
   (setq dir (expand-file-name dir))
   (if (file-directory-p dir)
(progn
  (if (and getsubdirs)
      (progn
 (setq buffer (get-buffer-create "*idltags*"))
 (call-process "sh" nil buffer nil "-c"
       (concat "find " dir " -type d -print"))
 (setq save_buffer (current-buffer))
 (set-buffer buffer)
 (setq files (idl-split-string
       (idl-replace-string
        (buffer-substring 1 (point-max))
        "\n" "/*.pro ")))
 (set-buffer save_buffer)
 (kill-buffer buffer))
    (setq files (list (concat dir "/*.pro"))))
  ;;
  ;; For each subdirectory
  (setq numfiles 0)
  (setq item (nth numfiles files))
  (while (and item)
    ;;
    ;; Call etags
    (if (not (string-match "^[ \\t]*$" item))
(progn
  (message (concat "Tagging " item "..."))
  (setq errbuf (get-buffer-create "*idltags-error*"))
  (setq status (+ status
    (call-process "sh" nil errbuf nil "-c"
    (concat cmd append item))))
  ;;
  ;; Append additional tags
  (setq append " --append ")
  (setq numfiles (1+ numfiles))
  (setq item (nth numfiles files)))
      (progn
 (setq numfiles (1+ numfiles))
 (setq item (nth numfiles files))
 )))

  (setq numdirs (1+ numdirs))
```

```
   (setq dir (nth numdirs directories)))
      (progn
 (setq numdirs (1+ numdirs))
 (setq dir (nth numdirs directories)))))

  (setq errbuf (get-buffer-create "*idltags-error*"))
  (if (= status 0)
      (kill-buffer errbuf))
  (message "")
  )


(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)))))

;; Additions for use with imenu.el (pop-up a list of IDL units in
;; the current file.

(defun idl-prev-index-position ()
"Search for the previous procedure or function.
Return nil if not found.  For use with imenu.el."
  (save-match-data
    (cond
     ((idl-find-key "\\<\\(pro\\|function\\)\\>" -1 'nomark))
      ;;   ((idl-find-key idl-begin-unit-reg 1 'nomark)
     (t nil))))

(defun idl-unit-name ()
  "Return the unit name.
```

```
Assumes that point is at the beginning of the unit as found by
`idl-prev-index-position'."
  (forward-sexp 2)
  (forward-sexp -1)
  (let ((begin (point)))
    (re-search-forward
"[a-zA-Z][a-zA-Z0-9$_]+\\(::[a-zA-Z][a-zA-Z0-9$_]+\\)?")
    (if (fboundp 'buffer-substring-no-properties)
 (buffer-substring-no-properties begin (point))
      (buffer-substring begin (point)))))

;; Menus - using easymenu.el
(defvar idl-mode-menus
  '("IDL"
    "--Movement--"
    ["Subprogram Start" idl-beginning-of-subprogram t]
    ["Subprogram End" idl-end-of-subprogram t]
    ["Up Block" idl-backward-up-block t]
    ["Skip Block Backward" idl-backward-block t]
    ["Skip Block Forward" idl-forward-block t]
    ["Down Block" idl-down-block t]
    "--Mark--"
    ["Mark Subprogram" idl-mark-subprogram t]
    ["Mark Block" idl-mark-block t]
    ["Mark Header" idl-mark-doclib t]
    "--Format--"
    ["Indent Subprogram" idl-indent-subprogram t]
    ["(Un)Comment Region" idl-toggle-comment-region "C-c \;"]
    ["Continue (Split) line" idl-split-line t]
    ["Toggle Auto Fill" idl-auto-fill-mode t]
    ("Templates"
     ["Case" idl-case t]
     ["For" idl-for t]
     ["Procedure" idl-procedure t]
     ["Function" idl-function t]
     ["Repeat" idl-repeat t]
     ["While" idl-while t]
     ["Doc Header" idl-doc-header t]
     ["Log" idl-doc-modification t])
    ["Generate IDL tags" idl-make-tags t]
    ["Start IDL shell" idl-shell t]))

(if (or (featurep 'easymenu) (load "easymenu" t))
    (easy-menu-define
     idl-menu idl-mode-map "IDL editing menus" idl-mode-menus))

(provide 'idl)
```

;;; idl.el ends here