Subject: Fix for IDL 5.0 breaking idl-shell
Posted by Mary Jo Brodzik on Thu, 03 Jul 1997 07:00:00 GMT
View Forum Message <> Reply to Message

This is a multi-part message in MIME format.

--------------167E2781446B
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Hello, everyone,

I have attached a new idl-shell.el (version 2.3), that fixes the
problem posted to this newsgroup last week, in which idl-shell was
incorrectly parsing the breakpoint message, and was opening a new
buffer instead of the correct buffer with the appropriate .pro
file to debug. Many thanks to Chris Chase for the fix!

Chris developed and until recently has been maintaining idl-shell,
but is no longer able to do so, having recently accepted a job with
no access to IDL. Chris asked me to post this fix, along with the
following information:

From Chris:

> A new variable `idl-shell-fix-inserted-breaks' and function
> `idl-shell-remove-breaks' were added. These fix up the original
> function `idl-shell-parse-line' which grabs the file, line number and
> module information. `idl-shell-file-line-message' is the regular
> expression that matches the necessary components. It has not changed.
> Instead, `idl-shell-remove-breaks' massages the output before the
> matching is done.
>
> This fix depends on the file name ending in ".pro" and that no ".pro"
> followed by whitespace appears previously to the end of the file name.
>
> So this fix is not robust, but it should work most of the time.
>

The fix also assumes that IDL 5.0 is inserting two spaces along with a
newline in the output message. If this changes at some time in the
future,
Chris has inserted an alternative solution that is clearly commented.

To use it, put the following in your .emacs or idl-shell-mode-hook:

> (setq idl-shell-fix-inserted-breaks t)
>

> This could be made the default. I think that it should work with IDL 4,
> but
> just in case I put the above switch in. If you decide it works well
> with both IDL 4 and 5 then just make 't' the default for
> `idl-shell-fix-inserted-breaks' in idl-shell.el.

From what Chris tells me, this latest idl-shell.el is now looking for
a volunteer to maintain it, but in the meantime, it is being archived
at Phil Williams' site, (Phil currently archives idl.el at this site,
and promised me he would make it visible there sometime next week), at

ftp://irc.chmcc.org/pub/idl_emacs

Sincerely,
Mary Jo

^^^
Mary Jo Brodzik
Scientific Programmer/Analyst
National Snow & Ice Data Center
Cooperative Institute for Research in
Environment Sciences
^^^

--------------167E2781446B
Content-Type: text/plain; charset=us-ascii; name="idl-shell.el"
Content-Transfer-Encoding: 7bit
Content-Disposition: inline; filename="idl-shell.el"

```
;;; idl-shell.el --- Run IDL as an inferior process of Emacs.
;;
;; No longer maintained or archived by original author Chris Chase
;;
;; Author: chase@jackson.jhuapl.edu (no longer valid)
;; Keywords: processes
;; Version: $Revision: 2.3 $ ($Date: 1996/06/30 18:27:00 $)
;;
;; LCD Archive Entry:
;; idl-shell|unknown|unknown|
;; Run an IDL process in an emacs buffer using comint.el|
;; $Date: 1996/06/30 18:27:00 $|$Revision: 2.3 $|~/misc/idl-shell.el.Z|
;;
;; 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:

;; Beginning with version 1.21, idl-shell.el should only be used with
;; IDL version 4 or greater.  You must use idl-shell.el version 1.20
;; with earlier versions of IDL.

;; Runs IDL as an inferior process of Emacs, much like the emacs
;; `shell' or `telnet' commands.  Provides command history and
;; searching.  Provides debugging commands available in buffers
;; visiting IDL procedure files, e.g., breakpoint setting, stepping,
;; execution until a certain line, printing expressions under point,
;; visual line pointer for current execution line, etc.
;;
;; When compiling a procedure, try using `idl-shell-save-and-compile',
;; \C-c\C-d\C-c, directly in an emacs ".PRO" file buffer rather than
;; entering .run directly (this avoids having to recompile the file
;; with the full path name when creating breakpoints).
;;
;; Can goto points of successive syntax errors from compilations -
;; `idl-shell-goto-next-error'.
;;
;; Requires idl.el and optionally uses easymenu.el for menus.
;;
;; This is essentially a beta version.  It could use some improvement.
;;
;; Uses comint.el.  Thus, only works with emacs version 19.  It may
;; work with emacs version 18 if you have comint.el installed - but I
;; haven't tried it.
;;
;; Could be used with PV-WAVE without the debugging commands.
;;
;; I suggest that you look at the info for comint. Here is some of the
;; stuff I extracted from comint.el.  Notice that many of the IDL line
;; editing keys are similar to the definitions preceded by "\C-c".
```

```
;;; Brief Command Documentation:
 ;;;============================================================
===================
;;; Comint Mode Commands: (common to all derived modes, like shell & cmulisp
;;; mode)
;;;
;;; m-p     comint-previous-input         Cycle backwards in input history
;;; m-n     comint-next-input           Cycle forwards
;;; m-r     comint-previous-matching-input  Previous input matching a regexp
;;; m-s     comint-next-matching-input      Next input that matches
;;; m-c-l   comint-show-output      Show last batch of process output
;;; return  comint-send-input
;;; c-d     comint-delchar-or-maybe-eof     Delete char unless at end of buff
;;; c-c c-a comint-bol                      Beginning of line; skip prompt
;;; c-c c-u comint-kill-input          ^u
;;; c-c c-w backward-kill-word         ^w
;;; c-c c-c comint-interrupt-subjob      ^c
;;; c-c c-z comint-stop-subjob          ^z
;;; c-c c-\ comint-quit-subjob          ^\
;;; c-c c-o comint-kill-output      Delete last batch of process output
;;; c-c c-r comint-show-output      Show last batch of process output
;;; c-c c-l comint-dynamic-list-input-ring  List input history
;;;

;; Installation:
;;
;;   To install put the following in your .emacs file (you may want to
;;   specify a complete path for the idl-shell.el file if it is
;;   not in a directory contained in `load-path'):
;;
;;  (autoload 'idl-shell "idl-shell" "Run IDL as an inferior process" t)
;;
;; Source:
;;
;;   The latest version can be obtained via anonymous FTP:
;;
;;   ftp://fermi.jhuapl.edu/pub/idl_emacs/idl-shell.el
;;   ftp://eos.crseo.ucsb.edu/pub/idl/idl-shell.el.Z
;;
;; Revision History:
;;
;; $Log: idl-shell.el,v $
;; Revision 2.3  1996/06/20  18:27:00  chase
;; (idl-shell-fix-inserted-breaks) - set to non-nil to run
;; `idl-shell-remove-breaks' which is now called by
;; idl-shell-parse-line.  This is a hack for IDLv5 which inserts line
;; breaks into messages messing up idl-shell.
;;
```

```
;;
;; Revision 2.2  1995/09/25  15:06:54  chase
;; (idl-shell-filter): Do not accumulate output into a separate string.
;; Instead grab it after seeing the prompt.  Requires putting hidden
;; output into a separate buffer.  Needs improvement to make more robust,
;; but works for now.
;;
;; Revision 2.1  1995/07/12  13:53:06  chase
;; Fixed bug due to change in easymenu.el with newer version of emacs.
;;
;; Revision 2.0  1995/07/10  13:41:23  chase
;; New release for IDL version 4.0 or higher only.  Many mods
;; specific to the changes in IDL 4 messages and in handling breakpoints.
;; In IDL 4 fixes some to the problems with breakpoints making some of
;; the previous workarounds unnecessary.
;;
;; Revision 1.21  1995/06/06  20:03:21  chase
;; Changed idl-shell-file-line-message to catch newlines.
;;
;; Revision 1.20  1995/05/03  14:38:09  chase
;; A temporary fix for X-Emacs (Lucid) which does not use overlays.
;;
;; Revision 1.19  1995/04/25  14:26:55  chase
;; (idl-shell-scan-for-state): `idl-shell-halt-frame' now contains the
;; IDL module in addition to the frame.  Should only use frame for
;; searching `idl-shell-bp-alist'.
;; (idl-shell-display-line): Reposition of buffer to syntax error.  Does
;; not work if the buffer with the error is currently selected.
;; (idl-shell-pc-frame): Use the halt frame only if is in the same file
;; and subroutine as the trace frame.  If the user executes a return
;; statement the halt frame can be inaccurate.  In this case use the
;; trace frame.
;; (idl-shell-parse-line): Adds current subroutine name to list.
;; (idl-shell-file-line-message): Added additional match for subroutine
;; name.
;;
;; Revision 1.18  1995/02/18  00:10:05  chase
;; Updated the LCD Archive Entry.
;;
;; Revision 1.17  1995/02/17  23:56:05  chase
;; (idl-shell-display-line): Added ability to display a face at the IDL
;; stop line.
;; (idl-shell-stop-line-face): Added.
;; (idl-shell-stop-line-overlay): Added.
;; (idl-shell-filter-bp): New version of IDL changed the formatting of
;; breakpoint listings.
;;
;; Revision 1.16  1995/01/26  16:58:29  chase
;; Added LCD Archive Entry.
;;
;; Revision 1.15  1994/10/26  23:13:17  chase
;; (idl-shell-clear-current-bp): Added a hack to delete breakpoints that
;; were originally set at the first statement of a block since IDL will
;; not break there.
;; (idl-shell-goto-next-error): Attempt to find the column where IDL
;; thinks the syntax error is.  Uses idl-shell-display-line.  Put in
;; debug keymap ("\C-n" ending).
;; (idl-shell-display-line): Added optional column argument for syntax
;; errors.
;; (idl-shell-scan-for-state): Shows compile error in other window.
;; (idl-shell-file-line-message): IDL file-line messages may be broken
;; across a line at the space before the line number.
;; (idl-shell-pc-frame): uses new variable `idl-shell-watch-for-bugs' to
;; catch potential bugs.
;; (idl-shell-new-bp): Check for replacing an existing breakpoint.
;; (idl-shell-set-bp): Don't classify pro|function statements as
;; begin-end blocks for setting breakpoints at the first line.
;;
;; Revision 1.14  1994/09/23  23:15:53  chase
;; (idl-shell-run-region): Added and placed on a key.
;;
;; Revision 1.13  1994/06/30  15:17:41  chase
;; Set IDL !more=0 in at startup (idl-shell).
;;
;; Revision 1.12  1994/06/29  01:34:03  chase
;; New FTP sites.
;;
;; Revision 1.11  1994/06/29  01:25:46  chase
;; Far too many changes to enumerate.  Changed internal breakpoint
;; formats.  Greatly improved tracking of breakpoints.  Better
;; determination of the actual statement where IDL execution is stopped.
;; Implementation of menus.  Display of syntax errors from compilations.
;;
;;
;; Known problems:
;;
;; The idl-shell buffer seems to occasionally lose output from the IDL
;; process.  I have not been able to consistently observe this.  I
;; do not know if it is a problem with idl-shell, comint, or Emacs
;; handling of subprocesses.
;;

;; I don't plan on implementing directory tracking by watching the IDL
;; commands entered at the prompt, since too often an IDL procedure
;; will change the current directory. If you want the the idl process
;; buffer to match the IDL current working just execute `M-x
;; idl-shell-resync-dirs' (bound to "\C-c\C-d\C-w" by default.)
;;
;; To Do:
;;
;; Explain the design of idl-shell.el
;;
;; Develop a protocol for determining if an IDL procedure needs to be
;; recompiled.  Now it is up to the user - use
;; idl-shell-save-and-compile (\C-c\C-d\C-c by default.)
;;

;;;   Customization variables

;; For customized settings, change these variables in your .emacs file
;; using `idl-shell-mode-hook'. For example:
;;
;;  (add-hook 'idl-shell-mode-hook
;; (function
;;  (lambda ()
;;    (setq   ; Set options here
;;          idl-shell-prompt-pattern "^WAVE> "  ; see def below
;;          idl-shell-overlay-arrow "=>"
;;          idl-shell-fix-inserted-breaks t
;;     )
;;    ;; Run other functions here
;;    )))
;;   ;;  These variables are used by `idl-shell' before the hook is run. So
;;   ;;  they must be set explicitly before calling `idl-shell'.
;;  (setq idl-shell-explicit-file-name "wave")
;;  (setq idl-shell-process-name "wave")

(require 'comint)
(require 'idl)

(defvar idl-shell-prompt-pattern "^IDL> "
  "*Regexp to match IDL prompt at beginning of a line.
For example, \"^IDL> \" or \"^WAVE> \".
The \"^\" means beginning of line.
This variable is used to initialise `comint-prompt-regexp' in the
process buffer.

This is a fine thing to set in your `.emacs' file.")

(defvar idl-shell-overlay-arrow ">"
  "*The overlay arrow to display at source lines where execution halts.")

(defvar idl-shell-stop-line-face 'underline
  "*The face for `idl-shell-stop-line-overlay'.
Allows you to choose the font, color and other properties for
line where IDL is stopped.")

(defvar idl-shell-fix-inserted-breaks nil
  "*If non-nil then run `idl-shell-remove-breaks' to clean up IDL messages.")

;;; Do not set this in the hook - that is run too late.  Explicitly
;;; set it in your .emacs.
(defvar idl-shell-explicit-file-name "idl"
  "*If non-nil, is the command to run IDL.
Should be an absolute file path or path relative to the current environment
execution search path.")

(defvar idl-shell-prefix-key "\C-c\C-d"
  "*The prefix key to contain the debugging map `idl-shell-mode-prefix-map'.")

(defvar idl-shell-process-name "idl"
  "*Name to be associated with the IDL process.  The buffer for the
process output is made by surrounding this name with `*'s.")

(defvar idl-shell-temp-pro-prefix "/tmp/idltemp"
  "*The prefix for temporary IDL files used when compiling regions.
It should be an absolute pathname.
The full temporary file name is obtained by to using `make-temp-name'
so that the name will be unique among multiple Emacs processes.")

(defvar idl-shell-mode-hook '()
  "*Hook for customising IDL-Shell mode.")

(defvar idl-shell-automatic-start nil
  "*If non-nil attempt invoke idl-shell if not already running.
This is checked when an attempt to send a command to an
IDL process is made.")

(defvar idl-shell-watch-for-bugs nil
  "*If non-nil Beeps with a message at potential idl-shell bugs.")

;;; End user customization variables

(defvar idl-shell-temp-pro-file nil
  "Absolute pathname for temporary IDL file for compiling regions")

(defvar idl-shell-dirstack-query "printd"
  "Command used by `idl-shell-resync-dirs' to query IDL for
the directory stack.")

(defvar idl-shell-stop-line-overlay nil
  "The overlay for where IDL is currently stopped.")
```

```
(if window-system
    (progn
      (if (not (fboundp 'overlayp))
   (defun overlayp (arg) nil))
 (setq idl-shell-stop-line-overlay (make-overlay 0 0))
 (delete-overlay idl-shell-stop-line-overlay))))

(defvar idl-shell-bp-query "help,/breakpoints"
  "Command to obtain list of breakpoints")

(defvar idl-shell-mode-map '())
(defvar idl-shell-mode-prefix-map '())

(if idl-shell-mode-map
    ()
  (setq idl-shell-mode-map (copy-keymap comint-mode-map))
  (define-key idl-shell-mode-map "\M-?" 'comint-dynamic-list-completions)
  (define-key idl-shell-mode-map "\t" 'comint-dynamic-complete))

(if idl-shell-mode-prefix-map
    ()
  (define-prefix-command 'idl-shell-mode-prefix-map)
  (define-key idl-shell-mode-prefix-map "\C-w" 'idl-shell-resync-dirs)
  (define-key idl-shell-mode-prefix-map "\C-b" 'idl-shell-break-here)
  (define-key idl-shell-mode-prefix-map "\C-h" 'idl-shell-to-here)
  (define-key idl-shell-mode-prefix-map "\C-e" 'idl-shell-run-region)
  (define-key idl-shell-mode-prefix-map "\C-r" 'idl-shell-cont)
  (define-key idl-shell-mode-prefix-map "\C-p" 'idl-shell-print)
  (define-key idl-shell-mode-prefix-map "\C-s" 'idl-shell-step)
  (define-key idl-shell-mode-prefix-map "\C-l" 'idl-shell-redisplay)
  (define-key idl-shell-mode-prefix-map "\C-u" 'idl-shell-up)
  (define-key idl-shell-mode-prefix-map "\C-c" 'idl-shell-save-and-compile)
  (define-key idl-shell-mode-prefix-map "\C-o" 'idl-shell-out)
  (define-key idl-shell-mode-prefix-map "\C-n" 'idl-shell-stepover)
  (define-key idl-shell-mode-prefix-map "\C-x" 'idl-shell-goto-next-error)
  (define-key idl-shell-mode-prefix-map "\C-d" 'idl-shell-clear-current-bp))


(defvar idl-shell-command-output nil
  "String for accumulating current command output.")

(defvar idl-shell-post-command-hook nil
  "Lisp list expression or function to run when an IDL command is finished.
The current command is finished when the IDL prompt is displayed.
This is evaluated if it is a list or called with funcall.")

(defvar idl-shell-hide-output nil
  "If non-nil the process output is not inserted into the output
  buffer.")

(defvar idl-shell-accumulation nil
  "Accumulate last line of output.")

(defvar idl-shell-pending-commands nil
  "List of commands to be sent to IDL.
Each element of the list is list of \(CMD PCMD HIDE\), where CMD is a
string to be sent to IDL and PCMD is a post-command to be placed on
`idl-shell-post-command-hook'. If HIDE is non-nil, hide the output
from command CMD. PCMD and HIDE are optional.")

(defun idl-shell-buffer ()
  "Name of buffer associated with IDL process.
The name of the buffer is made by surrounding `idl-shell-process-name
with `*'s."
  (concat "*" idl-shell-process-name "*"))

(defvar idl-shell-ready nil
  "If non-nil can send next command to IDL process.")

(defun idl-shell-mode ()
  "Major mode for interacting with an inferior IDL process.
Return after the end of the process' output sends the text from the
end of process to the end of the current line.  Return before end of
process output copies the current line (except for the prompt) to the
end of the buffer and sends it.

TAB in the middle of a file name attempts to complete the filename -
granted that the IDL process and emacs IDL buffer working directories
are synchronized.

The commands bound to the prefix key idl-shell-mode-prefix-map
\\[idl-shell-mode-prefix-map], are used for debugging code, e.g.,
setting and deleteing breakpoints, stepping, jumping out of blocks or
procedures, evaluating expressions, etc.  Get help for the
corresponding key, \"C-hk\".  When IDL is halted in the middle of a
procedure, the corresponding line of that procedure file is displayed
with an overlay arrow in anonther window.  The debugging commands are
available on the same key bindings when in an idl-mode buffer.

Command history, searching of previous commands, command line editing
are available via the comint-mode key bindings, by default mostly on
the key \"C-c\".

idl-shell-resync-dirs \\[idl-shell-resync-dirs] queries IDL in order to change Emacs current
    directory to correspond to the IDL process current directory.
```

```
\\{idl-shell-mode-map}
Customization: Entry to this mode runs the hooks on `comint-mode-hook' and
`idl-shell-mode-hook' (in that order)."
  (interactive)
  (setq comint-prompt-regexp idl-shell-prompt-pattern)
  (setq comint-process-echoes t)
  ;; Can not use history expansion because "!" is used for system variables.
  (setq comint-input-autoexpand nil)
  (setq comint-input-ring-size 64)
  (make-local-variable 'comint-completion-addsuffix)
  (setq comint-completion-addsuffix '("/" . ""))
  (setq comint-input-ignoredups t)
  (setq major-mode 'idl-shell-mode)
  (setq mode-name "IDL-Shell")
;;  (make-local-variable 'idl-shell-bp-alist)
  (setq idl-shell-halt-frame nil
 idl-shell-trace-frame nil
 idl-shell-command-output nil
 idl-shell-step-frame nil)
  (idl-shell-display-line nil)
  ;; Make sure comint-last-input-end does not go to beginning of
  ;; buffer (in case there were other processes already in this buffer).
  (set-marker comint-last-input-end (point))
  (setq idl-shell-ready nil)
  (setq idl-shell-bp-alist nil)
  (setq idl-shell-post-command-output nil)
  (setq idl-shell-sources-alist nil)
;;  (make-local-variable 'idl-shell-temp-pro-file)
  (setq idl-shell-hide-output nil
 idl-shell-temp-pro-file
 (concat (make-temp-name idl-shell-temp-pro-prefix) ".pro"))
  (easy-menu-add idl-shell-menu idl-shell-mode-map)
  (use-local-map idl-shell-mode-map)
  (fset 'idl-debug-map idl-shell-mode-prefix-map)
  (run-hooks 'idl-shell-mode-hook)
  (idl-shell-send-command idl-shell-initial-commands nil 'hide)
  (define-key idl-shell-mode-map idl-shell-prefix-key 'idl-shell-mode-prefix-map))

(defvar idl-shell-initial-commands "!more=0"
  "Initial commands, separated by newlines, to send to IDL.
This string is sent to the IDL process by `idl-shell-mode' which is
invoked by `idl-shell'.")

(defun idl-shell ()
  "Run an inferior IDL, with I/O through buffer `idl-shell-buffer'.
If buffer exists but shell process is not running, start new IDL.
If buffer exists and shell process is running,
 just switch to buffer `idl-shell-buffer'.
Program used comes from variable `idl-shell-explicit-file-name'.

The buffer is put in IDL-Shell mode, giving commands for sending input
and controlling the IDL job.  See help on `idl-shell-mode'.
See also the variable `idl-shell-prompt-pattern'.

\(Type \\[describe-mode] in the shell buffer for a list of commands.)"
  (interactive)
  (if (not (comint-check-proc (idl-shell-buffer)))
  (let* ((prog (or idl-shell-explicit-file-name
     "idl")))
    (set-process-filter
     (get-buffer-process (make-comint idl-shell-process-name prog))
     'idl-shell-filter)
    (easy-menu-remove idl-menu)
    (easy-menu-add idl-menu idl-mode-map)
    (set-buffer (idl-shell-buffer))
    (idl-shell-mode)))
  (switch-to-buffer (idl-shell-buffer)))

(defun idl-shell-send-command (&optional cmd pcmd hide preempt)
  "Send a command to IDL process.

\(CMD PCMD HIDE\) are placed at the end of `idl-shell-pending-commands'.
If IDL is ready the first command, CMD, in
`idl-shell-pending-commands' is sent to the IDL process.  If optional
second argument PCMD is non-nil it will be placed on
`idl-shell-post-command-hook' when CMD is executed.  If the optional
third argument HIDE is non-nil, then hide output from CMD.
If optional fourth argument PREEMPT is non-nil CMD is put at front of
`idl-shell-pending-commands'.

IDL is considered ready if the prompt is present
and if `idl-shell-ready' is non-nil."
  (save-excursion
    (let ((proc (get-buffer-process (set-buffer (idl-shell-buffer)))))
      (if (not proc)
   ;; No IDL process
   (if idl-shell-automatic-start
       ;; Try to start IDL process
       (progn
  (idl-shell)
  (setq proc (get-buffer-process (set-buffer (idl-shell-buffer))))
  (if (not proc)
      (error "Unable to start/find IDL process.")))
     (error "No IDL process currently running")))
      (goto-char (process-mark proc))
      ;; To make this easy, always push CMD onto pending commands
      (if cmd
   (setq idl-shell-pending-commands
  (if preempt
      ;; Put at front.
      (append (list (list cmd pcmd hide))
       idl-shell-pending-commands)
    ;; Put at end.
    (append idl-shell-pending-commands
     (list (list cmd pcmd hide))))))
      ;; Check if IDL ready
      (if (and idl-shell-ready
        ;; Check for IDL prompt
        (save-excursion
   (beginning-of-line)
   (looking-at idl-shell-prompt-pattern)))
   ;; IDL ready for command
   (if idl-shell-pending-commands
       ;; execute command
       (let* ((lcmd (car idl-shell-pending-commands))
       (pcmd (nth 1 lcmd))
       (hide (nth 2 lcmd)))
  ;; Set post-command
  (setq idl-shell-post-command-hook pcmd)
  ;; Output hiding
;;; Debug code
;;;  (setq idl-shell-hide-output nil)
  (setq idl-shell-hide-output hide)
  ;; Pop command
  (setq idl-shell-pending-commands
        (cdr idl-shell-pending-commands))
  ;; Send command for execution
  (set-marker comint-last-input-start (point))
  (set-marker comint-last-input-end (point))
  (comint-simple-send proc (car lcmd))
  (setq idl-shell-ready nil)))))))

;; There was a report that a newer version of comint.el changed the
;; name of comint-filter to comint-output-filter.  Unfortunately, we
;; have yet to upgrade.

(if (fboundp 'comint-output-filter)
    (fset 'idl-shell-comint-filter (symbol-function 'comint-output-filter))
  (fset 'idl-shell-comint-filter (symbol-function 'comint-filter)))

(defun idl-shell-filter (proc string)
  "Replace Carriage returns in output. Watch for prompt.
When the IDL prompt is received executes `idl-shell-post-command-hook'
```

```
and then calls `idl-shell-send-command' for any pending commands.
Also check for termination of process and cleanup."
  (if (not (eq (process-status idl-shell-process-name) 'run))
      (idl-shell-cleanup)
    (let ((data (match-data)))
      (unwind-protect
   (progn
     ;; May change the original match data.
     (let (p)
       (while (setq p (string-match "\C-M" string))
  (aset string p ?  )))
;;; Test/Debug code
;;     (save-excursion (set-buffer (get-buffer-create "*test*"))
;;       (goto-char (point-max))
;;       (insert "%%%" string))
     ;;
     ;; Keep output

; Should not keep output because the concat is costly.  If hidden put
; the output in a hide-buffer.  Then when the output is needed in post
; processing can access either the hide buffer or the idl-shell
; buffer.  Then watching for the prompt is easier.  Furthermore, if it
; is hidden and there is no post command, could throw away output.
;     (setq idl-shell-command-output
;    (concat idl-shell-command-output string))
     ;; Insert the string. Do this before getting the
     ;; state.
     (if idl-shell-hide-output
  (save-excursion
    (set-buffer
     (get-buffer-create "*idl-shell-hidden-output*"))
    (goto-char (point-max))
    (insert string))
       (idl-shell-comint-filter proc string))
     ;; Watch for prompt - need to accumulate the current line
     ;; since it may not be sent all at once.
     (if (string-match "\n" string)
  (setq idl-shell-accumulation
        (substring string
     (progn (string-match "\\(.*\n\\)*" string)
     (match-end 0))))
       (setq idl-shell-accumulation
      (concat idl-shell-accumulation string)))
     ;; Check for prompt in current line
     (if (setq idl-shell-ready
        (string-match idl-shell-prompt-pattern
        idl-shell-accumulation))
  (progn
    (if idl-shell-hide-output
        (save-excursion
   (set-buffer "*idl-shell-hidden-output*")
   (goto-char (point-min))
   (re-search-forward idl-shell-prompt-pattern nil t)
   (setq idl-shell-command-output
         (buffer-substring (point-min) (point)))
   (delete-region (point-min) (point)))
      (setq idl-shell-command-output
     (save-excursion
       (set-buffer
        (process-buffer proc))
       (buffer-substring
        (progn
          (goto-char (process-mark proc))
          (beginning-of-line nil)
          (point))
        comint-last-input-end))))
;;; Test/Debug code
;;    (save-excursion (set-buffer
;;       (get-buffer-create "*idl-shell-output*"))
;;      (goto-char (point-max))
;;      (insert "%%%" string))
    ;; Scan for state and do post command - bracket them
    ;; with idl-shell-ready=nil since they
    ;; may call idl-shell-send-command.
    (let ((idl-shell-ready nil))
      (idl-shell-scan-for-state)
      ;; Unset idl-shell-ready to prevent sending
      ;; commands to IDL while running hook.
      (if (listp idl-shell-post-command-hook)
   (eval idl-shell-post-command-hook)
        (funcall idl-shell-post-command-hook))
      ;; Reset to default state for next command.
      ;; Also we do not want to find this prompt again.
      (setq idl-shell-accumulation nil
     idl-shell-command-output nil
     idl-shell-post-command-hook nil
     idl-shell-hide-output nil))
    ;; Done with post command. Do pending command if
    ;; any.
    (idl-shell-send-command))))
 (store-match-data data)))))

(defun idl-shell-scan-for-state ()
  "Scan for state info.
Looks for messages in output from last IDL command indicating where
IDL has stopped. The types of messages we are interested in are
```

```
execution halted, stepped, breakpoint, interrupted at and trace
messages.  We ignore error messages otherwise.
For breakpoint messages process any attached count or command
parameters.
Update the windows if a message is found."
  (let (update)
    (cond
     ;; Make sure we have output
     ((not idl-shell-command-output))
     ;; Various types of HALT messages.
     ((string-match
       (mapconcat 'identity idl-shell-halt-messages "\\|")
       idl-shell-command-output)
      ;; Grab the file and line state info.
      (setq idl-shell-halt-frame
     (idl-shell-parse-line
      (substring idl-shell-command-output (match-end 0)))
     update t))
     ;; Handle breakpoints separately
     ((string-match idl-shell-break-message
      idl-shell-command-output)
      (setq idl-shell-halt-frame
     (idl-shell-parse-line
      (substring idl-shell-command-output (match-end 0)))
     update t)
      ;; Do breakpoint count and command processing
      (let ((bp (assoc
   (list
    (nth 0 idl-shell-halt-frame)
    (nth 1 idl-shell-halt-frame))
   idl-shell-bp-alist)))
 (if bp
     (let ((count (idl-shell-bp-get bp 'count))
    (cmd (idl-shell-bp-get bp 'cmd)))
       ;; no longer need to use count
;       (if count
;    (if (= (setq count (1- count)) 0)
;        ;; Delete breakpoint
;        (idl-shell-clear-bp bp)
;      ;; Update breakpoint in alist
;      (idl-shell-set-bp-data bp (list count cmd))))
       (if cmd
    ;; Execute command
    (if (listp cmd)
        (eval cmd)
      (funcall cmd))))
   ;; A breakpoint that we did not know about - perhaps it was
   ;; set by the user or IDL isn't reporting breakpoints like
   ;; we expect.
   (idl-shell-bp-query)))))
    ;;; Handle compilation errors in addition to the above
    (if (and idl-shell-command-output
      (string-match
       idl-shell-syntax-error idl-shell-command-output))
 (save-excursion
   (set-buffer
    (get-buffer-create idl-shell-error-buffer))
   (erase-buffer)
   (insert idl-shell-command-output)
   (goto-char (point-min))
   (setq idl-shell-error-last (point))
   (idl-shell-goto-next-error)))
    ;; Do update
    (if update
 (idl-shell-display-line (idl-shell-pc-frame)))))

(defvar idl-shell-error-buffer
  "*idl-shell-errors*"
  "Buffer containing syntax errors from IDL compilations.")

(defvar idl-shell-syntax-error
  "^% Syntax error.\n\\s-*At:\\s-*\\(.*\\),\\s-*Line\\s-*\\(.*\\)"
  "A regular expression to match an IDL syntax error.
The first \(..\) pair should match the file name.  The second pair
should match the line number.")

(defvar idl-shell-file-line-message
  "\\<\\([^\n\t ]+\\)\\>[\n\t ]+\\<\\([0-9]+\\)\\>[\n\t ]+\\([^\n\t ]+\\)\\>"
  "*A regular expression to parse out the file name and line number.
The first \(..\) pair should match the subroutine name.  The third
\(..\) pair should match the file name. The second pair should match
the line number.")

(defun idl-shell-parse-line (string-in)
  "Parse IDL message for the subroutine, file and line number."
  (let ((string (idl-shell-remove-breaks string-in)))
    (if (string-match idl-shell-file-line-message string)
 (list
  (save-match-data
    (file-truename (substring string (match-beginning 3) (match-end 3))))
  (string-to-int
   (substring string (match-beginning 2) (match-end 2)))
  (substring string (match-beginning 1) (match-end 1))))))

;;; This function relies on IDL 5 inserting a 3 characters (a return
;;; and two spaces) and a single '.pro' appearing at the end of the
;;; filename ('.pro' followed by whitespace anywhere else might mess
;;; this up).  Not very robust (nothing in idl-shell.el is) but it works.
(defun idl-shell-remove-breaks (string)
  "IDL v5 may split the messages in the middle of words.
We search for '.pro' string and remove all intervening
returns followed by 2 spaces."
  (if (not idl-shell-fix-inserted-breaks)
      string
    (save-excursion
      (set-buffer (get-buffer-create idl-shell-bp-buffer))
      (erase-buffer)
      (insert string)
      (goto-char (point-min))
      (while (not (let ((eol (progn (end-of-line) (point))))
      (beginning-of-line)
      (or (re-search-forward "\\.pro\\s-*$" eol 'limit)
   (eobp))))
 ;; Delete the newline and two spaces.  If it isn't
 ;; consistently two spaces then try
 ;; (delete-horizontal-space) instead, but this could
 ;; cause other problems like merging the linenumber and
 ;; filename if the break falls between them (which is
 ;; only likely if the module name is very long).
 (forward-char 1)
 ;; (delete-horizontal-space)
 (if (looking-at "  ")
     (delete-char 2))
 (delete-char -1))
      (buffer-substring (point-min) (point-max)))))

;;; The following are the types of messages we attempt to catch to
;;; resync our idea of where IDL execution currently is.
;;;

(defvar idl-shell-halt-frame nil
  "The frame associated with halt/breakpoint messages.")

(defvar idl-shell-step-frame nil
  "The frame associated with step messages.")

(defvar idl-shell-trace-frame nil
  "The frame associated with trace messages.")

(defconst idl-shell-halt-messages
  '("^% Execution halted at"
    "^% Interrupted at:"
    "^% Stepped to:"
    "^% At "
    "^% Stop encountered:"
    )
  "*A list of regular expressions matching IDL messages.
These are the messages containing file and line information where
IDL is currently stopped.")

(defconst idl-shell-trace-messages
  '("^% At " ;; First line of a trace message
    )
  "*A list of regular expressions matching IDL trace messages.
These are the messages containing file and line information where
IDL will begin looking for the next statement to execute.")

(defconst idl-shell-step-messages
  '("^% Stepped to:"
    )
  "*A list of regular expressions matching stepped execution messages.
These are IDL messages containing file and line information where
IDL has currently stepped.")

(defvar idl-shell-break-message "^% Breakpoint at:"
  "*Regular expression matching an IDL breakpoint message line.")

(defun idl-shell-cleanup ()
  "Do necessary cleanup for a terminated IDL process."
  ;; Remove overlay
  (setq idl-shell-step-frame nil
 idl-shell-halt-frame nil
 idl-shell-pending-commands nil)
  (idl-shell-display-line nil))

(defun idl-shell-resync-dirs ()
  "Resync the buffer's idea of the current directory stack.
This command queries IDL with the command bound to
`idl-shell-dirstack-query' (default \"printd\"), reads the
output for the new directory stack."
  (interactive)
  (idl-shell-send-command idl-shell-dirstack-query
     'idl-shell-filter-directory
     'hide))

(defun idl-shell-filter-directory ()
  "Get the current directory from `idl-shell-command-output'.
Change the default directory for the process buffer to concur."
  (save-excursion
    (set-buffer (idl-shell-buffer))
    (if (string-match "Current Directory: *\\(\\S-*\\) *$" idl-shell-command-output)
 (cd (substring idl-shell-command-output (match-beginning 1) (match-end 1))))))
```

```
;;;
;;; This section cotains code for debugging IDL programs.
;;;

(defun idl-shell-redisplay (&optional hide)
  "Tries to resync the display with where execution has stopped.
Issues a \"help,/trace\" command followed by a call to
`idl-shell-display-line'."
  (interactive)
  (idl-shell-send-command
   "help,/trace"
   '(idl-shell-display-line
     (idl-shell-pc-frame))
     hide))

(defun idl-shell-goto-frame (&optional frame)
  "Set buffer to FRAME with point at the frame line.
If the optional argument FRAME is nil then idl-shell-pc-frame is
used.  Does nothing if the resulting frame is nil."
  (if frame ()
    (setq frame (idl-shell-pc-frame)))
  (cond
   (frame
    (set-buffer (find-file-noselect (car frame)))
    (widen)
    (goto-line (nth 1 frame)))))

(defun idl-shell-pc-frame ()
  "Returns the frame for IDL execution."
  (and idl-shell-halt-frame
       (list (nth 0 idl-shell-halt-frame) (nth 1 idl-shell-halt-frame))))

(defun idl-shell-valid-frame (frame)
  "Check that frame is for an existing file."
  (file-readable-p (car frame)))

(defun idl-shell-display-line (frame &optional col)
  "Display FRAME file in other window with overlay arrow.

FRAME is a list of file name, line number, and subroutine name.
If FRAME is nil then remove overlay."
  (if (not frame)
      ;; Remove stop-line overlay
      (progn
 (setq overlay-arrow-string nil)
 (if (overlayp idl-shell-stop-line-overlay)
     (delete-overlay idl-shell-stop-line-overlay)))
    (if (not (idl-shell-valid-frame frame))
 (error (concat "Invalid frame - unable to access file: " (car frame)))
      (let* ((buffer (find-file-noselect (car frame)))
      (shell-window (get-buffer-window (idl-shell-buffer)))
      (select-shell (equal (buffer-name) (idl-shell-buffer)))
      window
      pos)
 ;; First select IDL shell window
 (if shell-window
     (select-window shell-window)
   (switch-to-buffer (idl-shell-buffer)))
 ;; Select file in other window
 (setq window (display-buffer buffer 'not-this-window))
 (save-excursion
   (set-buffer buffer)
   (save-restriction
     (widen)
     (goto-line (nth 1 frame))
     (setq pos (point))
     (if (overlayp idl-shell-stop-line-overlay)
  ;; Make overlay
  (progn
    (overlay-put
     idl-shell-stop-line-overlay 'face idl-shell-stop-line-face)
    (move-overlay
     idl-shell-stop-line-overlay
     (point) (save-excursion (end-of-line) (point))
     (current-buffer)))
       (setq overlay-arrow-string idl-shell-overlay-arrow)
       (or overlay-arrow-position
    (setq overlay-arrow-position (make-marker)))
       (set-marker overlay-arrow-position (point) buffer)))
   (cond ((or (< pos (point-min)) (> pos (point-max)))
   (widen)
   (goto-char pos)))
   (if col (move-to-column col))
   (setq pos (point)))
 (set-window-point window pos)
 ;; (or (pos-visible-in-window-p pos window)
 ;;    (window-height window ))
 ;;      (redraw-frame (window-frame window))
 (if (and (equal (buffer-name) (idl-shell-buffer)) (not select-shell))
     (select-window window))))))

(defun idl-shell-step (arg)
  "Step one source line. If given prefix argument ARG, step ARG source lines."
  (interactive "p")
  (or (not arg) (< arg 1)
    (setq arg 1))
  (idl-shell-send-command (concat ".s " arg)))

(defun idl-shell-stepover (arg)
  "Stepover one source line.
If given prefix argument ARG, step ARG source lines.
Uses IDL's stepover executive command which does not enter called functions."
  (interactive "p")
  (or (not arg) (< arg 1)
    (setq arg 1))
  (idl-shell-send-command (concat ".so " arg)))

(defun idl-shell-break-here (count &optional cmd)
  "Set breakpoint at current line.

If Count is nil then an ordinary breakpoint is set.  We treat a count
of 1 as a temporary breakpoint using the ONCE keyword.  Counts greater
than 1 use the IDL AFTER=count keyword to break only after reaching
the statement count times.

Optional argument CMD is a list or function to evaluate upon reaching
the breakpoint."

  (interactive "P")
  (if (listp count)
      (setq count nil))
  (idl-shell-set-bp
   ;; Create breakpoint
   (idl-shell-bp (idl-shell-current-frame)
   (list count cmd)
   (idl-shell-current-module))))

(defun idl-shell-set-bp-check (bp)
  "Check for failure to set breakpoint.
This is run on `idl-shell-post-command-hook'.
Offers to recompile the procedure if we failed.  This usually fixes
the problem with not being able to set the breakpoint."
  ;; Scan for message
  (if (and idl-shell-command-output
    (string-match "% BREAKPOINT: *Unable to find code"
    idl-shell-command-output))
      ;; Offer to recompile
      (progn
 (if (progn
       (beep)
       (y-or-n-p
        (concat "Okay to recompile file "
         (idl-shell-bp-get bp 'file) " ")))
```

```
     ;; Recompile
     (progn
       ;; Clean up before retrying
       (idl-shell-command-failure)
       (idl-shell-send-command
        (concat ".run " (idl-shell-bp-get bp 'file)) nil nil)
       ;; Try setting breakpoint again
       (idl-shell-set-bp bp))
   (beep)
   (message "Unable to set breakpoint.")
   (idl-shell-command-failure)
   )
 ;; return non-nil if no error found
 nil)
    'okay))

(defun idl-shell-command-failure ()
  "Do any necessary clean up when an IDL command fails.
Call this from a function attached to `idl-shell-post-command-hook'
that detects the failure of a command.
For example, this is called from `idl-shell-set-bp-check' when a
breakpoint can not be set."
  ;; Clear pending commands
  (setq idl-shell-pending-commands nil))

(defun idl-shell-cont ()
  "Continue executing."
  (interactive)
  (idl-shell-send-command ".c" '(idl-shell-redisplay 'hide)))

(defun idl-shell-clear-bp (bp)
  "Clear breakpoint BP.
Clears in IDL and in `idl-shell-bp-alist'."
  (let ((index (idl-shell-bp-get bp)))
    (if index
 (progn
   (idl-shell-send-command
    (concat "breakpoint,/clear," index))
   (idl-shell-bp-query)))))

(defun idl-shell-current-frame ()
"Return a list containing the current file name and line point is in.
If in the IDL shell buffer, returns `idl-shell-pc-frame'."
  (if (eq (current-buffer) (get-buffer (idl-shell-buffer)))
      ;; In IDL shell
      (idl-shell-pc-frame)
    ;; In source
    (list (file-truename (buffer-file-name))
   (save-restriction
     (widen)
     (save-excursion
       (beginning-of-line)
       (1+ (count-lines 1 (point)))))))

(defun idl-shell-current-module ()
  "Return the name of the module for the current file.
Returns nil if unable to obtain a module name."
  (if (eq (current-buffer) (get-buffer (idl-shell-buffer)))
      ;; In IDL shell
      (nth 2 idl-shell-halt-frame)
    ;; In pro file
    (save-restriction
      (widen)
      (save-excursion
 (if (idl-prev-index-position)
     (upcase (idl-unit-name)))))))

(defun idl-shell-clear-current-bp ()
  "Remove breakpoint at current line."
  (interactive)
  (let ((bp (idl-shell-find-bp (idl-shell-current-frame))))
    (if bp (idl-shell-clear-bp bp)
      ;; Try moving to beginning of statement
      (save-excursion
 (idl-shell-goto-frame)
 (idl-beginning-of-statement)
 (setq bp (idl-shell-find-bp (idl-shell-current-frame)))
 (if bp (idl-shell-clear-bp bp)
   (beep)
   (message "Cannot identify breakpoint for this line"))))))

(defun idl-shell-to-here ()
  "Set a breakpoint with count 1 then continue."
  (interactive)
  (idl-shell-break-here 1)
  (idl-shell-cont))


(defun idl-shell-up ()
  "Run to end of current block."
  (interactive)
  (if (idl-shell-pc-frame)
      (save-excursion
 (idl-shell-goto-frame)
 ;; find end of subprogram
 (let ((eos (save-excursion
       (idl-beginning-of-subprogram)
       (idl-forward-block)
       (point))))
   (idl-backward-up-block -1)
   ;; move beyond end block line - IDL will not break there.
   ;; That is, you can put a breakpoint there but when IDL does
   ;; break it will report that it is at the next line.
   (idl-next-statement)
   (idl-end-of-statement)
   ;; Make sure we are not beyond subprogram
   (if (< (point) eos)
       ;; okay
       ()
     ;; Move back inside subprogram
     (goto-char eos)
     (idl-previous-statement))
   (idl-shell-to-here)))))

(defun idl-shell-out ()
  "Attempt to run until this procedure exits.
Runs to the last statement and then steps 1 statement."
  (interactive)
  (idl-shell-send-command (concat ".o")))

(defun idl-shell-print ()
  "Print current expression.
An expression is an identifier plus 1 pair of matched parentheses
directly following the identifier - an array or function
call. Alternatively, an expression is the contents of any matched
parentheses when the open parentheses is not directly preceded by an
identifier. If point is at the beginning or within an expression
return the inner-most containing expression, otherwise, return the
preceding expression."
  (interactive)
  (save-excursion
    (let (beg
   end
   (name "[a-zA-Z][a-zA-Z0-9$_]*"))
      ;; Move to beginning of current or previous expression
      (if (looking-at "\\<\\|(")
   ;; At beginning of expression, don't move backwards unless
   ;; this is at the end of an indentifier.
   (if (looking-at "\\>")
       (backward-sexp))
 (backward-sexp))
      (if (looking-at "\\>")
   ;; Move to beginning of identifier - must be an array or
   ;; function expression.
```

```
   (backward-sexp))
      ;; Move to end of expression
      (setq beg (point))
      (forward-sexp)
      (while (looking-at "\\>(\\|\\.")
 ;; an array
 (forward-sexp))
      (setq end (point))
      (idl-shell-send-command
       (concat "print," (buffer-substring beg end))))))

(defvar idl-shell-bp-alist nil
  "Alist of breakpoints.
A breakpoint is a cons cell \(\(file line\) . \(\(index module\) data\)\)

The car is the frame for the breakpoint:
file - full path file name.
line - line number of breakpoint - integer.

The first element of the cdr is a list of internal IDL data:
index - the index number of the breakpoint internal to IDL.
module - the module for breakpoint internal to IDL.

Remaining elements of the cdr:
data - Data associated with the breakpoint by idl-shell currently
contains two items:

count - number of times to execute breakpoint. When count reaches 0
the breakpoint is cleared and removed from the alist.
command - command to execute when breakpoint is reached, either a
lisp function to be called with `funcall' with no arguments or a
list to be evaluated with `eval'.")

(defun idl-shell-run-region (beg end &optional n)
  "Compile and run the region using the IDL process.
Copies the region to a temporary file `idl-shell-temp-pro-file'
and issues the IDL .run command for the file.  Because the
region is compiled and run as a main program there is no
problem with begin-end blocks extending over multiple
lines - which would be a problem if `idl-shell-evaluate-region'
was used.  An END statement is appended to the region if necessary.

If there is a prefix argument, display IDL process."
  (interactive "r\nP")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (find-file-noselect idl-shell-temp-pro-file))
      (erase-buffer)
      (insert-buffer-substring oldbuf beg end)
      (if (not (save-excursion
   (idl-previous-statement)
   (idl-look-at "\\<end\\>")))
   (insert "\nend\n"))
      (save-buffer 0)))
  (idl-shell-send-command (concat ".run " idl-shell-temp-pro-file))
  (if n
      (display-buffer (idl-shell-buffer))))

(defun idl-shell-evaluate-region (beg end &optional n)
  "Send region to the IDL process.
If there is a prefix argument, display IDL process."
  (interactive "r\nP")
  (idl-shell-send-command (buffer-substring beg end))
  (if n
      (display-buffer (idl-shell-buffer))))

(defvar idl-shell-bp-buffer "*idl-shell-bp*"
  "Scratch buffer for parsing IDL breakpoint lists and other stuff.")

(defun idl-shell-bp-query ()
  "Reconcile idl-shell's breakpoint list with IDL's.
Queries IDL using the string in `idl-shell-bp-query'."
  (interactive)
  (idl-shell-send-command idl-shell-bp-query
     'idl-shell-filter-bp
     'hide))

(defun idl-shell-bp-get (bp &optional item)
  "Get a value for a breakpoint.
BP has the form of elements in idl-shell-bp-alist.
Optional second arg ITEM is the particular value to retrieve.
ITEM can be 'file, 'line, 'index, 'module, 'count, 'cmd, or 'data.
'data returns a list of 'count and 'cmd.
Defaults to 'index."
  (cond
   ;; Frame
   ((eq item 'line) (nth 1 (car bp)))
   ((eq item 'file) (nth 0 (car bp)))
   ;; idl-shell breakpoint data
   ((eq item 'data) (cdr (cdr bp)))
   ((eq item 'count) (nth 0 (cdr (cdr bp))))
   ((eq item 'cmd) (nth 1 (cdr (cdr bp))))
   ;; IDL breakpoint info
   ((eq item 'module) (nth 1 (car (cdr bp))))
   ;;    index - default
   (t (nth 0 (car (cdr bp))))))
```

```
(defun idl-shell-filter-bp ()
  "Get the breakpoints from `idl-shell-command-output'.
Create `idl-shell-bp-alist' updating breakpoint count and command data
from previous breakpoint list."
  (save-excursion
    (set-buffer (get-buffer-create idl-shell-bp-buffer))
    (erase-buffer)
    (insert idl-shell-command-output)
    (goto-char (point-min))
    (let ((old-bp-alist idl-shell-bp-alist))
      (setq idl-shell-bp-alist (list nil))
      (if (re-search-forward "^\\s-*Index.*\n\\s-*-" nil t)
   (while (and
    (not (progn (forward-line) (eobp)))
    ;; Parse breakpoint line.
    ;; Breakpoints have the form:
    ;;  Index Module Line File
    ;;  All seperated by whitespace.
    ;;
    ;;  Add the breakpoint info to the list
    (re-search-forward
     " \\s-*\\(\\S-+\\)\\s-+\\(\\S-+\\)\\s-+\\(\\S-+\\)\\s-+\\(\\S- +\\) " nil t))
     (nconc idl-shell-bp-alist
     (list
      (cons
       (list
        (save-match-data
   (file-truename
    (buffer-substring ; file
     (match-beginning 4) (match-end 4))))
        (string-to-int ; line
         (buffer-substring
   (match-beginning 3) (match-end 3))))
       (list
        (list
         (buffer-substring ; index
   (match-beginning 1) (match-end 1))
         (buffer-substring ; module
   (match-beginning 2) (match-end 2)))
        ;; idl-shell data: count, command
        nil nil))))))
      (setq idl-shell-bp-alist (cdr idl-shell-bp-alist))
      ;; Update count, commands of breakpoints
      (mapcar 'idl-shell-update-bp old-bp-alist)))
  idl-shell-bp-alist)

(defun idl-shell-update-bp (bp)
  "Update BP data in breakpoint list.
If BP frame is in `idl-shell-bp-alist' updates the breakpoint data."
  (let ((match (assoc (car bp) idl-shell-bp-alist)))
    (if match (setcdr (cdr match) (cdr (cdr bp))))))

(defun idl-shell-set-bp-data (bp data)
  "Set the data of BP to DATA."
  (setcdr (cdr bp) data))

(defun idl-shell-bp (frame &optional data module)
  "Create a breakpoint structure containing FRAME and DATA.  Second
and third args, DATA and MODULE, are optional.  Returns a breakpoint
of the format used in `idl-shell-bp-alist'.  Can be used in commands
attempting match a breakpoint in `idl-shell-bp-alist'."
  (cons frame (cons (list nil module) data)))

(defvar idl-shell-old-bp nil
  "List of breakpoints previous to setting a new breakpoint.")

(defun idl-shell-sources-bp (bp)
  "Check `idl-shell-sources-alist' for source of breakpointusing BP.
If an equivalency is found, return the IDL internal source name.
Otherwise return the filename in bp."
  (let*
      ((bp-file (idl-shell-bp-get bp 'file))
       (bp-module (idl-shell-bp-get bp 'module))
       (internal-file-list (cdr (assoc bp-module idl-shell-sources-alist))))
    (if (and
  internal-file-list
  (equal bp-file (nth 0 internal-file-list)))
 (nth 1 internal-file-list)
      bp-file)))

(defun idl-shell-set-bp (bp)
  "Try to set a breakpoint BP.

The breakpoint will be placed at the beginning of the statement on the
line specified by BP or at the next IDL statement if that line is not
a statement.
Determines IDL's internal representation for the breakpoint which may
have occured at a different line then used with the breakpoint
command."

  ;; Get and save the old breakpoints
  (idl-shell-send-command idl-shell-bp-query
     '(progn
        (idl-shell-filter-bp)
        (setq idl-shell-old-bp idl-shell-bp-alist))
    'hide)
  ;; Get sources for IDL compiled procedures followed by setting
  ;; breakpoint.
  (idl-shell-send-command idl-shell-sources-query
    (` (progn
         (idl-shell-sources-filter)
         (idl-shell-set-bp2 (quote (, bp)))))
    'hide))

(defun idl-shell-set-bp2 (bp)
  "Use results of breakpoint and sources query to set bp.
Use the count argument with IDLs breakpoint command.
We treat a count of 1 as a temporary breakpoint.
Counts greater than 1 use the IDL AFTER=count keyword to break
only after reaching the statement count times."
  (let*
      ((arg (idl-shell-bp-get bp 'count))
       (key (cond
      ((not (and arg (numberp arg))) "")
      ((= arg 1)
       ",/once")
      ((> arg 1)
       (format ",after=%d" arg)))))
    (idl-shell-send-command
     (concat "breakpoint,'" (idl-shell-sources-bp bp) "',"
      (idl-shell-bp-get bp 'line)
      key)
     ;; Check for failure and look for breakpoint in IDL's list
     (` (progn
   (if (idl-shell-set-bp-check (quote (, bp)))
       (idl-shell-set-bp3 (quote (, bp)))))
 )
     ;; do not hide output
     nil
     'preempt)))

(defun idl-shell-set-bp3 (bp)
  "Find the breakpoint in IDL's internal list of breakpoints."
  (idl-shell-send-command idl-shell-bp-query
    (` (progn
         (idl-shell-filter-bp)
         (idl-shell-new-bp (quote (, bp)))))
    'hide
    'preempt))

(defun idl-shell-find-bp (frame)
  "Return breakpoint from `idl-shell-bp-alist' for frame.
Returns nil if frame not found."
```

```
  (assoc frame idl-shell-bp-alist))

(defun idl-shell-new-bp (bp)
  "Find the new breakpoint in IDL's list and update with DATA.
The actual line number for a breakpoint in IDL may be different than
the line number used with the IDL breakpoint command.
Looks for a new breakpoint index number in the list.  This is
considered the new breakpoint if the file name of frame matches."
  (let ((obp-index (mapcar 'idl-shell-bp-get idl-shell-old-bp))
 (bpl idl-shell-bp-alist))
    (while (and (member (idl-shell-bp-get (car bpl)) obp-index)
  (setq bpl (cdr bpl))))
    (if (and
  (not bpl)
  ;; No additional breakpoint.
  ;; Need to check if we are just replacing a breakpoint.
  (setq bpl (assoc (car bp) idl-shell-bp-alist)))
 (setq bpl (list bpl)))
    (if (and bpl
      (equal (idl-shell-bp-get (setq bpl (car bpl)) 'file)
      (idl-shell-bp-get bp 'file)))
 ;; Got the breakpoint - add count, command to it.
 ;; This updates `idl-shell-bp-alist' because a deep copy was
 ;; not done for bpl.
 (idl-shell-set-bp-data bpl (idl-shell-bp-get bp 'data))
      (beep)
      (message "Failed to identify breakpoint in IDL"))))

(defun idl-shell-save-and-compile ()
  "Save file and compile it in IDL.
Runs `save-buffer', \\[save-buffer], and then sends a '.RUN' command for
the associated file to IDL."
  (interactive)
  (save-buffer)
  (idl-shell-send-command
   (concat ".run " (buffer-file-name)) nil nil)
  (idl-shell-bp-query))

(defvar idl-shell-sources-query "help,/source"
  "IDL command to obtain source files for compiled procedures.")

(defvar idl-shell-sources-alist nil
  "Alist of IDL procedure names and compiled source files.
Elements of the alist have the form:

  (module name . (source-file-truename IDL-internal-filename)).")

(defun idl-shell-sources-query ()
  "Determine source files for IDL compiled procedures.
Queries IDL using the string in `idl-shell-sources-query'."
  (interactive)
  (idl-shell-send-command idl-shell-sources-query
     'idl-shell-sources-filter
     'hide))

(defun idl-shell-sources-filter ()
  "Get source files from `idl-shell-sources-query' output.
Create `idl-shell-sources-alist' consisting of
list elements of the form:
 (module name . (source-file-truename IDL-internal-filename))."
  (save-excursion
    (set-buffer (get-buffer-create idl-shell-bp-buffer))
    (erase-buffer)
    (insert idl-shell-command-output)
    (goto-char (point-min))
    (let (cpro cfun)
      (if (re-search-forward "Compiled Procedures:" nil t)
   (progn
     (forward-line) ; Skip $MAIN$
     (setq cpro (point))))
      (if (re-search-forward "Compiled Functions:" nil t)
   (progn
     (setq cfun (point))
     (setq idl-shell-sources-alist
    (append
     ;; compiled procedures
     (progn
       (beginning-of-line)
       (narrow-to-region cpro (point))
       (goto-char (point-min))
       (idl-shell-sources-grep))
     ;; compiled functions
     (progn
       (widen)
       (goto-char cfun)
       (idl-shell-sources-grep)))))))))

(defun idl-shell-sources-grep ()
  (save-excursion
    (let ((al (list nil)))
      (while (and
       (not (progn (forward-line) (eobp)))
       (re-search-forward
        "\\s-*\\(\\S-+\\)\\s-+\\(\\S-+\\)" nil t))
 (nconc al
        (list
  (cons
   (buffer-substring ; name
    (match-beginning 1) (match-end 1))
   (let ((internal-filename
   (buffer-substring ; source
    (match-beginning 2) (match-end 2))))
     (list
      (file-truename internal-filename)
      internal-filename))
      ))))
      (cdr al))))


(defun idl-shell-clear-all-bp ()
  "Remove all breakpoints in IDL."
  (interactive)
  (idl-shell-send-command
   idl-shell-bp-query
   '(progn
      (idl-shell-filter-bp)
      (mapcar 'idl-shell-clear-bp idl-shell-bp-alist))
   'hide))

(defvar idl-shell-error-last 0
  "Position of last syntax error in `idl-shell-error-buffer'.")

(defun idl-shell-goto-next-error ()
  "Move point to next IDL syntax error."
  (interactive)
  (let (frame col)
    (save-excursion
      (set-buffer idl-shell-error-buffer)
      (goto-char idl-shell-error-last)
      (if (re-search-forward idl-shell-syntax-error nil t)
   (progn
     (setq frame
    (list
     (save-match-data
       (file-truename
        (buffer-substring (match-beginning 1) (match-end 1))))
     (string-to-int
      (buffer-substring (match-beginning 2)
          (match-end 2)))))
     ;; Try to find the column of the error
     (save-excursion
       (setq col
      (if (re-search-backward "\\^" nil t)
   (current-column)
```

```
        0)))))
       (setq idl-shell-error-last (point)))

     (if frame
  (progn
    (idl-shell-display-line frame col))
       ;;   (pop-to-buffer idl-shell-error-buffer)
       ;;   (goto-char idl-shell-error-last)
       ;;   (recenter 4)
       ;;   (switch-to-buffer-other-window (find-file-noselect (car frame)))
       ;;   (goto-line (nth 1 frame))
       ;;   (move-to-column col))
       (beep)
       (message "No more errors."))))

;; Menus - using easymenu.el
(defvar idl-shell-file-menus
  '(["Set breakpoint" idl-shell-break-here t]
    ["Clear breakpoint" idl-shell-clear-current-bp t]
    ["Go to Here" idl-shell-to-here t]
    ["Save & Compile" idl-shell-save-and-compile t]
    ["Continue" idl-shell-cont t]
    ["Step" idl-shell-step t]
    ["Stepover" idl-shell-stepover t]
    ["Run Region" idl-shell-run-region t]
    ["Show Next Compile Error" idl-shell-goto-next-error t]))

(defvar idl-shell-menus
  '("Debug"
    ["Get Working Directory" idl-shell-resync-dirs t]
    ["Clear All Breakpoints" idl-shell-clear-all-bp t]
    ["Redisplay" idl-shell-redisplay t]))

(if (or (featurep 'easymenu) (load "easymenu" t))
    (progn
      (easy-menu-define
       idl-menu idl-mode-map "IDL editing menus"
       (append idl-mode-menus (list "--Debug--") idl-shell-file-menus))
      (easy-menu-define
       idl-shell-menu idl-shell-mode-map "IDL shell menus"
       (append idl-shell-menus idl-shell-file-menus))))


(provide 'idl-shell)

--------------167E2781446B--
```