An Emacs Mode for Aldor

Ralf Hemmecke

November 12, 2005

Contents

1 Introduction

Aldor (www.aldor.org) is a programming language that has its origin in the computer algebra system AXIOM and is particularly tailored for symbolic computation.

The source files are ordinary text files and can be edited by any editor. We provide an EMACS mode for this language with support of syntax highlighting, support for automatically indenting lines, and support for sending lines from a source file to a buffer where the aldor interpreter runs.

My former implementation of `aldor.el` was built on the `cc-mode`. Unfortunately, it did only work in XEMACS but not in EMACS. First, I wanted to rebuild `aldor-mode` on a newer version of `cc-mode`. However, I soon realised that due major syntactic differences between Aldor and C-like languages such an approach would need too much programming.

Fortunately, Tobias Beck wrote a first version of `aldor-mode` from scratch and thus demonstrated to me how short indentation support could be implemented in elisp without building on `cc-mode`. I took his code as a starting base but actually reprogrammed everything to reach a greater flexibility and extensibility. Only code that provides support for sending Aldor commands to a buffer with a running Aldor interpreter is nearly taken literally from Tobias. So acknowledgement goes to him. Thank you Tobias.

2 Usage

2.1 Configuration files

The `aldor-mode` can be used in the following way. Put the following lines into your `.emacs` file and store `aldor.el` into some place where Emacs can find it. For example, into `~/pub/elisp/aldor.el` (see `init.el`).

You should say

```
notangle -R.emacs aldor.el.nw > .emacs
```

or

```
notangle -Rcustom.el aldor.el.nw > .xemacs/custom.el
notangle -Rinit.el   aldor.el.nw > .xemacs/init.el
```

depending on whether you use EMACS or XEMACS.

Of course you should not simply override these files but somehow add the contents to the files you already have.

Where for EMACS there is only one file `.emacs`, XEMACS splits the initialisation into two parts which are usually stored under the `.xemacs` directory.

⟨*.emacs*⟩≡
  ⟨*init.el*⟩
  ⟨*custom.el*⟩

The following code is just a sample of how one can load `aldor-mode` automatically when loading a `.as` file.

⟨*init.el*⟩≡
```
(setq load-path (append '("~/pub/elisp") load-path))
(require 'font-lock)
(autoload 'aldor-mode "aldor" "Aldor Mode" t)
(add-hook 'aldor-mode-hook
          '(lambda ()
             (font-lock-mode t)
             (font-lock-fontify-buffer)))
(setq auto-mode-alist
      (append '(("\\.as$" . aldor-mode)) auto-mode-alist))
```

Customisation of EMACS is done by customising variables and faces. In XEMACS one can change them via the Options menu entry.

⟨*custom.el*⟩≡
```
⟨custom-set-variables⟩
⟨custom-set-faces⟩
```

We switch on `lazy-shot-mode`, because for big programs it will only fontlock the part of the file you actually see.

⟨*custom-set-variables*⟩≡
```
(custom-set-variables
 '(column-number-mode t)
 '(line-number-mode t)
 '(lazy-shot-mode t nil (lazy-shot))
)
```

We don't use filladapt at the moment, since we don't write ++ and – comments.

⟨*old-custom-set-variables*⟩≡
```
(custom-set-variables
 '(column-number-mode t)
 '(line-number-mode t)
 '(lazy-shot-mode t nil (lazy-shot))
 ⟨filladapt⟩
)
```

2.2 Using `filladapt`

I find it convenient to use the package filladapt. This correctly formats lines with comment characters continued (M-q).

⟨*filladapt*⟩≡
```
'(filladapt-mode t t (filladapt))
'(filladapt-mode-line-string " Fadap")
⟨filladapt-token-table⟩
⟨filladapt-token-conversion-table⟩
⟨filladapt-token-match-table⟩
```

To be honest, I cannot explain what the acutal entries mean, since I played around with them a long time ago, but for me they work and so I am to lazy to re-read the documentation of filladapt. I think the most important thing is the regular expressions for `aldor-comment` and `aldor-doc` in the following table.

⟨*filladapt-token-table*⟩≡
```
'(filladapt-token-table (quote (
    ("^" beginning-of-line)
    (">+" citation->)
    ("[A-Za-z0-9][^'‘\"<
]*>[    ]*" supercite-citation)
    ("[#%!;]*-+[a-z#%!;:]* " rhx-comment)
    ("-+ " aldor-comment)
    ("\\++ " aldor-doc)
    (";+" lisp-comment)
    ("#+" sh-comment)
    ("%+" postscript-comment)
    ("///*" c++-comment)
    ("@c[       ]" texinfo-comment)
    ("@comment[         ]" texinfo-comment)
    ("\\\\item[         ]" bullet)
    ("[0-9]+\\.[        ]" bullet)
    ("[0-9]+\\(\\.[0-9]+\\)+[   ]" bullet)
    ("[A-Za-z]\\.[      ]" bullet)
    ("(?[0-9]+)[        ]" bullet)
    ("(?[A-Za-z])[      ]" bullet)
    ("[0-9]+[A-Za-z]\\.[        ]" bullet)
    ("(?[0-9]+[A-Za-z])[        ]" bullet)
    ("[~*+]+[   ]" bullet)
    ("o[        ]" bullet)
    ("[         ]+" space)
    ("$" end-of-line))))
```

⟨*filladapt-token-conversion-table*⟩≡
```
'(filladapt-token-conversion-table (quote (
    (citation-> . exact)
    (supercite-citation . exact)
    (rhx-comment . exact)
    (aldor-comment . exact)
    (aldor-doc . exact)
    (lisp-comment . exact)
    (sh-comment . exact)
    (postscript-comment . exact)
    (c++-comment . exact)
    (texinfo-comment . exact)
    (bullet . spaces)
    (space . exact)
    (end-of-line . exact))))
```

⟨*filladapt-token-match-table*⟩≡
```
'(filladapt-token-match-table (quote (
    (citation-> citation->)
    (supercite-citation supercite-citation)
    (rhx-comment rhx-comment)
    (aldor-comment aldor-comment)
    (aldor-doc aldor-doc)
    (lisp-comment lisp-comment)
    (sh-comment sh-comment)
    (postscript-comment postscript-comment)
    (c++-comment c++-comment)
    (texinfo-comment texinfo-comment)
    (bullet)
    (space bullet space)
    (beginning-of-line beginning-of-line))))
```

2.3 Customising `font-lock`

These are my personal settings. I like to have a dark background and white text. If you do not like it this way then change it.

⟨*custom-set-faces*⟩≡
```
(custom-set-faces
  ⟨default-faces⟩
  ⟨font-lock-faces for default⟩
)
```

⟨*default-faces*⟩≡
```
'(default ((t (:foreground "white" :background "#005"))) t)
'(whitespace-blank-face ((t (:background "black"))))
'(paren-match ((t (:foreground "black" :background "darkseagreen2"))) t)
'(zmacs-region ((t (:background "black"))) t)
```

⟨*font-lock-faces for default*⟩≡
```
'(font-lock-string-face        ((t (:foreground "#a0a0ff"))))
'(font-lock-reference-face     ((t (:foreground "red"))))
'(font-lock-variable-name-face ((t (:foreground "coral"))))
'(font-lock-keyword-face       ((t (:foreground "SpringGreen"))))
'(font-lock-type-face          ((t (:foreground "orangered"))))
'(font-lock-comment-face       ((t (:foreground "Goldenrod"))))
'(font-lock-function-name-face ((t (:foreground "magenta"))))
;'(aldor-font-lock-preprocessor-face  ((t (:foreground "MediumAquamarine"))))
;'(aldor-font-lock-doc-face     ((t (:foreground "#a0a0ff"))))
;'(aldor-font-lock-throw-face   ((t (:foreground "red"))))
;'(aldor-font-lock-pretend-face ((t (:foreground "red"))))
;'(aldor-font-lock-assert-face  ((t (:foreground "red"))))
;'(aldor-font-lock-assert-argument-face  ((t (:foreground "yellow"))))
```

3 The `aldor-mode` Code

The code is split in mainly 3 sections

- Aldor loop code (mainly due to Tobias Beck),
- font-lock code for syntax highlighting,
- indentation support.

Finally, the `aldor-mode-setup` puts it all together.

⟨*⟩≡
```
⟨Header⟩
(provide 'aldor)
⟨Aldor loop⟩
⟨Syntax table⟩
⟨Syntax highlighting⟩
⟨Keyboard bindings⟩
⟨Indentation support⟩
⟨aldor-mode-setup⟩
```

Well, every file should have a header...

⟨*Header*⟩≡
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This file provides an Aldor mode for EMACS.
;; Starting date: 18-Jun-2004
;; $Id: aldor.el.nw,v 1.7 2005/07/05 16:17:37 hemmecke Exp $
;; Author: Ralf Hemmecke <ralf@hemmecke.de>
;; http://www.hemmecke.de
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Since the original file is written using a literate programming
;; style through the noweb facility of Norman Ramsey, documentation
;; of the code can be found in aldor.el.nw.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

3.1 Sending Command String to an Aldor Process

By means of the following code it is possible to start an interactive Aldor loop (`aldor -gloop`) by just pressing `C-RET` (control enter).

Furthermore, you can send a line from a file in an `aldor-mode` buffer to the interactive Aldor loop buffer by moving the cursor to this line and pressing `S-RET` (shift enter).

This section describes only the actual code that does the job. The binding of the keys is described in Section 3.4.

⟨*Aldor loop*⟩≡
⟨*Setup Aldor compiler and options*⟩
⟨*Make* `aldor-loop` *an interactive command*⟩
⟨*Provide elisp function to sent an Aldor command*⟩

We first set up the path to the Aldor compiler. We assume here that the environment variable `ALDORROOT` is set to point to Aldor.

⟨*Setup Aldor compiler and options*⟩≡
```
(require 'comint)
(defvar aldor-path (substitute-in-file-name "$ALDORROOT/bin/aldor"))
(defvar aldor-loop-switch "-gloop")
```

Now we make `aldor-loop` available for the user. The following code will be bound to `C-RET` (control enter) in Section 3.4. So typing `C-RET` should switch to just another window (EMACS-speak) and display the interactive Aldor loop there. If there is currently only one window, then it is split vertically (2 windows, one on top of the other) and the interactive Aldor loop is shown in one of them.

⟨*Make* `aldor-loop` *an interactive command*⟩≡
```
(defun aldor-loop ()
  "Run Aldor interpreter loop in a buffer."
  (interactive)
  (if (= (count-windows) 1)
      (split-window-vertically)
    (other-window 1))
  (switch-to-buffer
   (make-comint "Aldor" aldor-path nil aldor-loop-switch))
  (setq comint-prompt-regexp "^[^>]*[>]+ *"))
```

Finally, we provide a command that sends one line from an `aldor-mode` buffer to the interactive Aldor process. This code is almost taken literally from Tobias Beck. I have, however, changes some parts at the end so that the interactive Aldor window shows always the end of the buffer.

⟨*Provide elisp function to sent an Aldor command*⟩≡
```
(defun aldor-send-to-loop ()
  "Send one line of input to Aldor process."
  (interactive)
  (let* ((s (current-buffer))                    ; current buffer
         (bol (progn (beginning-of-line) (point)))
         (eol (progn (end-of-line) (point)))
         (cmd (buffer-substring bol eol)))     ; current line
    ; forward cursor one line in this buffer
    (forward-line)
    ; copy line to Aldor buffer
    (set-buffer "*Aldor*")
    (goto-char (point-max))
    (insert cmd)
    (goto-char (point-max))
    ; send input to Aldor interpreter
    (comint-send-input)
    (goto-char (point-max))
    (set-window-point (get-buffer-window "*Aldor*") (point))))
```

3.2 Syntax Table

We modify the syntax table specific for Aldor. Unfortunately it does not work to make `++` into a comment, so the `+` is not treated in any special way but rather becomes a character like `=`.

In particular, we treat the underscore as an escape character.

⟨*Syntax table*⟩≡

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

(if aldor-mode-syntax-table
    ()
  (let ((table (make-syntax-table)))
    (modify-syntax-entry ?- ". 12"   table)
    (modify-syntax-entry ?\r ">  "  table)
    (modify-syntax-entry ?\n ">  "  table)
    (modify-syntax-entry ?\t "    "  table)
    (modify-syntax-entry ?\\ ".   "  table)
    (modify-syntax-entry ?+  ".   "  table)
    (modify-syntax-entry ?*  ".   "  table)
    (modify-syntax-entry ?/  ".   "  table)
    (modify-syntax-entry ?=  ".   "  table)
    (modify-syntax-entry ?<  ".   "  table)
    (modify-syntax-entry ?>  ".   "  table)
    (modify-syntax-entry ?#  ".   "  table)
    (modify-syntax-entry ?_  "\\  "  table)
    (modify-syntax-entry ?%  "w   "  table)
    (modify-syntax-entry ?!  "w   "  table)
    (modify-syntax-entry ??  "w   "  table)
    (modify-syntax-entry ?\" "\"  "  table)
    (modify-syntax-entry ?\{ "(}  "  table)
    (modify-syntax-entry ?\[ "(]  "  table)
    (modify-syntax-entry ?\( "()  "  table)
    (modify-syntax-entry ?\} "){  "  table)
    (modify-syntax-entry ?\] ")[  "  table)
    (modify-syntax-entry ?\) ")(  "  table)
    (setq aldor-mode-syntax-table table)))
```

3.3 Syntax Highlighting

Syntax highlighting is built on the standard EMACS `font-lock` package. We first provide a list that connects the Aldor keywords to the various available abstract font-lock faces like, for example, `font-lock-variable-comment-face`. We define additional faces for special things like `throw`, `pretend`, and `assert`.

For compatibility reasons between Emacs and XEmacs, we also introduce faces for `++` and Aldor compiler system commands like `#if`.

⟨*Syntax highlighting*⟩≡
⟨*aldor-font-lock-faces*⟩
⟨*regular expression for a Aldor identifier (chars 2..n)*⟩
⟨*font-lock-keywords*⟩

Before we are going to describe the actual font-lock keywords, let us provide some extra faces for Aldor.

We must define `aldor-font-lock-doc-face` and `aldor-font-lock-preprocessor-face`, because of incompatibilities between Emacs and XEmacs. Thus we cannot take `font-lock-doc-face` or `font-lock-doc-string-face`. And the `font-lock-preprocessor-face` does not exist in Emacs.

For some misterious reason in Emacs, we have to use the `defvar` expressions below. They are unnecessary in XEmacs, but do not hurt.

⟨*aldor-font-lock-faces*⟩≡

```
(defface aldor-font-lock-throw-face
  '((((class color) (background dark)) (:foreground "red"))
    (((class color) (background light)) (:foreground "red"))
    (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
    (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
    (t (:bold t)))
  "Face for the Aldor throw keyword")
(defvar aldor-font-lock-throw-face 'aldor-font-lock-throw-face
  "Face for the Aldor throw keyword")

(defface aldor-font-lock-pretend-face
  '((((class color) (background dark)) (:foreground "red"))
    (((class color) (background light)) (:foreground "red"))
    (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
    (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
    (t (:bold t)))
  "Face for the Aldor pretend keyword")
(defvar aldor-font-lock-pretend-face 'aldor-font-lock-pretend-face
  "Face for the Aldor pretend keyword")

(defface aldor-font-lock-assert-face
  '((((class color) (background dark)) (:foreground "red"))
    (((class color) (background light)) (:foreground "red"))
    (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
    (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
    (t (:bold t)))
  "Face for the Aldor assert keyword")
(defvar aldor-font-lock-assert-face 'aldor-font-lock-assert-face
  "Face for the Aldor assert keyword")

(defface aldor-font-lock-assert-argument-face
  '((((class color) (background dark)) (:background "red"))
    (((class color) (background light)) (:background "red"))
    (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
    (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
    (t (:bold t)))
```

```
    "Face for the Aldor assert arguments")
  (defvar aldor-font-lock-assert-argument-face
    'aldor-font-lock-assert-argument-face
    "Face for the Aldor assert arguments")

  (defface aldor-font-lock-doc-face
    '((((class color) (background dark)) (:foreground "#a0a0ff"))
      (((class color) (background light)) (:foreground "#a0a0ff"))
      (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
      (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
      (t (:bold t)))
    "Face for the Aldor ++ comments")
  (defvar aldor-font-lock-doc-face
    'aldor-font-lock-doc-face
    "Face for the Aldor ++ comments")

  (defface aldor-font-lock-preprocessor-face
    '((((class color) (background dark)) (:foreground "MediumAquamarine"))
      (((class color) (background light)) (:foreground "MediumAquamarine"))
      (((class grayscale) (background light)) (:foreground "DimGray" :italic t))
      (((class grayscale) (background dark)) (:foreground "LightGray" :italic t))
      (t (:bold t)))
    "Face for the Aldor #if, #assert, #else etc.")
  (defvar aldor-font-lock-preprocessor-face
    'aldor-font-lock-preprocessor-face
    "Face for the Aldor #if, #assert, #else etc.")
```

For `aldor-font-lock-keywords` we often need to describe the characters of a Aldor identifier that can come after the first character. We put this regular expression here into an extra variable since we want to include the treatment of the underscore character.

⟨*regular expression for a Aldor identifier (chars 2..n)*⟩≡

```
  (defconst id-rest "\\(\\sw\\|_.\\)*")
  (defconst id-fun (concat "\\(\\<[a-z]" id-rest "\\>\\)"))
  (defconst id-dom (concat "\\(\\<[A-Z]" id-rest "\\>\\)"))
```

Now, let us describe the actual keywords that we use here. We group them into several sections. Some Aldor keywords are treated in a particular way like, for example,

```
import from D1, D2;
```

in order to be able to highlight also the corresponding domain and package names.

Our highlighting code relies somewhat on some conventions of how the Aldor source file should look like and will probably break if these conventions are not matched. See at the corresponding font-lock expression below for a description of these conventions.

The order in the following list is important and cannot be changed arbitrarily.

⟨*font-lock-keywords*⟩≡

```
(defconst aldor-font-lock-keywords
  (list
   ⟨++ Documentation comments⟩
   ⟨Aldor system commands⟩
   ⟨from and to statements⟩
   ⟨assert statements⟩
   ⟨throw keyword⟩
   ⟨pretend keyword⟩
   ⟨highlight brackets⟩
   ⟨mapsto +->⟩
   ⟨define categories and extend domains⟩
   ⟨Domain and package definitions⟩
   ⟨Function and constant definitions⟩
   ⟨Aldor remaining keywords⟩
  )
  "Subdued level highlighting for Aldor modes."
)
```

The general way to specify one of the following faces is

```
("regular expression" (number face))
```

where the regular expression should contain subexpressions embraced by `\\(` and `\\)`. The number then refers to this subexpression and the corresponding string is set in the font-lock face `face`.

3.3.1 Documentation Comments

First of all comments starting with ++ and reaching to the end of the line are considered to be documentation and should appear in a documentation face.

We do not need to give code for ordinary `--` comments, since they are handled by default via our setting of the syntax table, see Section 3.2.

⟨++ *Documentation comments*⟩≡

```
'("\\([+][+].*\\)"
  (1 aldor-font-lock-doc-face t t))
```

3.3.2 Aldor System Commands

The following code covers Aldor system commands like

```
#include "calix"
#assert CalixConfig
#library ModLib "mod.ao"
#if Option
```

The # sign should always appear in the first column.

⟨*Aldor system commands*⟩≡

```
(list (concat "^\\(#\\)[ \t]*\\<\\("
        (mapconcat 'identity '(
           "include"
           "reinclude"
           "assert"
           "unassert"
           "if"
           "elseif"
           "else"
           "endif"
           "line"
           "error"
;          "pile"    -- I don't like it.
;          "endpile" -- I don't like it.
           "library"
           "includeDir"
           "libraryDir"
           "int"
           "quit"
        )  "\\|")
        "\\)\\>[ \t]*\\(\\sw*\\)")
  '(1 aldor-font-lock-preprocessor-face)
  '(2 aldor-font-lock-preprocessor-face)
  '(3 font-lock-variable-name-face)
)
```

3.3.3 Statments involving `from` and `to`

The code covers statements like

```
... from xxx,yyy;
... to xxx, yyy;
```

Note that by convention the closing semicolon must appear on the same line. The first semicolon that is seen terminates the highlighting.

Before `from` and `to` might appear `import`, `inline`, and `export` possibly being followed by a list of function signatures. These other keywords are highlighted by including them in the list of remaining Aldor keywords, see below.

⟨*from and to statements*⟩≡
```
'("\\(\\<from\\|to\\>\\)[ \t]+\\([^;]*\\);"
  (1 font-lock-keyword-face)
  (2 font-lock-variable-name-face))
```

3.3.4 The `assert` Keyword

I consider the `assert` keyword of Aldor to be important and thus highlight it in a special way. Whenenver you change colors, `assert` should catch your eye. That was my intention.

⟨*assert statements*⟩≡
```
'("\\<\\(assert\\)\\>\\(.*\\)"
  (1 aldor-font-lock-assert-face)
  (2 aldor-font-lock-assert-argument-face))
```

3.3.5 The `throw` Keyword

I consider the throw keyword as something exceptional so it should have another face.

⟨*throw keyword*⟩≡
```
'("\\<\\(throw\\)\\>"
  (1 aldor-font-lock-throw-face))
```

3.3.6 The `pretend` Keyword

Similarly one should really know what one is doing when using the pretend keyword. So let us give it a special face.

⟨*pretend keyword*⟩≡
```
'("\\<\\(pretend\\)\\>"
  (1 aldor-font-lock-pretend-face))
```

3.3.7 Highlight Brackets

I also like to highlight the parentheses, braces and brackets. Don't ask me why.

⟨*highlight brackets*⟩≡
```
'("\\([][{}()]\\)"
  (1 font-lock-keyword-face))
```

3.3.8 The +-> Operator

I also find it useful to see the mapsto `+->` operator in a keyword face.

⟨*mapsto* `+->`⟩≡
```
'("\\([+]->\\)"
  (1 font-lock-keyword-face))
```

3.3.9 Defining Categories and Extending Domains

Another convention concerns categories. They should be introduced by the `define` keyword at the beginning of a line and followed by an identifier that starts with a capital letter.

Similarly we treat extension of domains. But instead of `define` they are introduced by the `extend` keyword.

Category and domain names should consist exclusively of word characters as defined by the syntax table, see Section 3.2.

⟨*define categories and extend domains*⟩≡
```
(list (concat "^\\(define\\|extend\\)[ \t]+" id-dom)
  '(1 font-lock-keyword-face)
  '(2 font-lock-type-face))
```

3.3.10 Defining Domains

By convention, in domain and package definitions the name should start in the first column and should start with a capital letter and followed exclusively by word characters as defined by the syntax table, see Section 3.2.

⟨*Domain and package definitions*⟩≡
```
(list (concat "^" id-dom)
  '(1 font-lock-type-face))
```

3.3.11 Defining Functions and Constants

Functions and constants should start with a small letter as the first thing on the line (after some possible white space) and followed exclusively by word characters as defined by the syntax table, see Section 3.2. We do not bother to highlight function names like * or +.

A function name should immediately be followed by an open parenthesis. A constant should immediately be followed by a colon and a space. By convention the == should appear on the same line. If this is not possible, the line should end with an open parenthesis. The local keyword can optionally appear in front of the function or constant name.

⟨*Function and constant definitions*⟩≡
```
(list (concat "^[ \t]*\\(local\\)[ \t]+" id-fun ".*\\(==\\|($\\)")
  '(1 font-lock-keyword-face)
  '(2 font-lock-function-name-face))
(list (concat "^[ \t]*" id-fun ".*\\(==\\|($\\)")
  '(1 font-lock-function-name-face))
```

3.3.12 Remaining Aldor Keywords

Finally, the Aldor keywords that we have not yet dealt with should be highlighted.

⟨*Aldor remaining keywords*⟩≡
```
(cons (concat "\\<\\(" (regexp-opt '(
    "add"      "and"       "always"
;   "assert"
    "break"    "but"      "catch"     "default"
;   "define"   -- Handled in "define CATEGORY"
;   "delay"    -- reserved Aldor keyword
    "do"       "else"     "except"    "export"
;   "extend"   -- Handled in "extend DOMAIN"
;   "fix"      -- reserved Aldor keyword
    "for"      "fluid"    "free"
;   "from"     -- handled in [[from and to statements]]
    "generate"
;   "goto"     -- THIS SHOULD NOT APPEAR IN PROGRAMS!!!
    "has"      "if"       "import"    "in"     "inline"
;   "is"       "isnt"      -- reserved Aldor keywords
    "iterate"
;   "let"      -- reserved Aldor keyword
    "local"    "macro"    "never"     "not"    "of"    "or"
;   "pretend"  -- Handled separately
    "ref"      "repeat"  "return"
;   "rule"     -- reserved Aldor keyword
    "select"    "then"
;   "throw"    -- Handled separately
;   "to"       -- handled in [[from and to statements]]
    "try"      "where"    "while"     "with"   "yield"))  "\\)\\>")
  '(1 font-lock-keyword-face))
```

3.4 Keyboard Bindings

In `aldor-mode` the enter key (`RET`) gets special meaning. We define a new `key-map` for Aldor. Pressing `RET` puts the cursor into the next line and indents it to the appropriate column.

Pressing `C-RET` (control enter) starts an interactive Aldor interpreter in a buffer called `*Aldor*`.

Pressing `S-RET` (shift enter) in an `aldor-mode` buffer sends the command under the cursor to the `*Aldor*` buffer and executes it.

⟨*Keyboard bindings*⟩≡
```
⟨aldor-newline⟩
⟨keymap⟩
```

Here comes the code that is called when RET is pressed.

⟨*aldor-newline*⟩≡
```
(defun aldor-newline ()
  "Newline for Aldor major mode."
  (interactive)
  (newline)
  (aldor-indent-new-line))
```

The functions that are called when a certain key is pressed is defined through the following aldor-mode-map.

⟨*keymap*⟩≡
```
(defvar aldor-mode-map
 (let ((aldor-mode-map (make-keymap)))
  (define-key aldor-mode-map [return]           'aldor-newline)
  (define-key aldor-mode-map [(shift return)]   'aldor-send-to-loop)
  (define-key aldor-mode-map [(control return)] 'aldor-loop)
  aldor-mode-map)
 "Keymap for Aldor major mode.")
```

3.5 Indentation Support

The idea of our indentation code is as simple as this.

1. Go to the last non-empty line that is not a special line (basically a comment-only line).
2. Compute the indentation of this line.
3. Then scan through this line and add for any open and closing bracket a certain offset amount which is described by the variables below. Let's say this give a value i for the indentation.
4. If the line that should be indented is not one of the special lines, i. e.,
   - an empty line,
   - a line started by `++` or `--`,
   - a line with `#` in the first column,
   - a line started by a closing bracket,

   then i is the amount of indentation.
5. Otherwise, we add an appropriate offset amount to i and take this as the indentation.

At the moment we do not bother to enforce a correct pairing of opening and closing brackets.

Our indentation code is structured into three sections. First we describe customisable variables. Then some auxiliary functions. And, finally, a function that puts it all together to compute the indentation of the current line and indents this line.

⟨*Indentation support*⟩≡
  ⟨*Indentation default offsets*⟩
  ⟨*Indentation auxiliaries*⟩
  ⟨*Indentation computation*⟩

3.5.1 Indentation Variables

There are basically two variables that control the amount of indentation. One is connected to opening and closing brackets and the other to a treatment of special lines like comments, empty lines, and Aldor system commands.

⟨*Indentation default offsets*⟩≡
  ⟨*Indentation open and close offsets*⟩
  ⟨*Indentation special line offsets*⟩

The following variable describes opening and closing brackets and the corresponding offset amount. For the closing bracket the amount will be automatically negated.

⟨*Indentation open and close offsets*⟩≡
```
(defvar aldor-offset-pair-alist
  '(
    (("("   . ")"  ) . 4)
    (("\\[" . "\\]") . 4)
    (("{"   . "}"  ) . 8)
    )
  "Regular expressions together with corresponding offset amount.")
```

The following variable describes the offset amount that should be added to the calculated one in order to achieve the right indentation also in these cases.

Note that we have put here a value of `-1000` to yield a negative offset. Negative offsets will be truncated to 0 so that the corresponding line will start at the first column.

⟨*Indentation special line offsets*⟩≡
```
(defvar aldor-special-line-offset-alist
  '(
    ("^[+][+][+]+"       . -1000)   ;predoc in col 0
    ("^[ \t]+[+][+][+]+" . 0)       ;indented predoc
    ("^[ \t]+[+][+]+"    . +8)      ;indented postdoc
    ("^--+"              . -1000)   ;comment in col 0
    ("^[ \t]+--+"        . 0)       ;indented comment
    ("^#"                . -1000)   ;aldor system commands
    ("^[ \t]*$"          . -1000)   ;empty line
    )
  "Regular expressions for comments together with the offsets.")
```

3.5.2 Indentation Auxiliary Functions

⟨*Indentation auxiliaries*⟩≡
⟨*Indentation helpful offset list*⟩
⟨*Indentation identify special and closing-bracket lines*⟩
⟨*Indentation forward-char skipping Aldor escape character*⟩
⟨*Indentation move the point behind the string*⟩

We compute this variable because it allows easier code for the function `aldor-compute-next-indentation`. It basically returns a list of pairs where each pair consist of a regular expression and an offset amount. The regular expressions for the closing brackets will also be contained in this list but with the corresponding negative offset amount.

⟨*Indentation helpful offset list*⟩≡
```
(defvar aldor-offset-alist
  (let* ((alist nil) (q nil) (p (reverse aldor-offset-pair-alist)))
    (while p
      (setq q (car p))
      (setq p (cdr p))
      (setq alist (cons (cons (cdar q) (- (cdr q))) alist))
      (setq alist (cons (cons (caar q) (cdr q))     alist)))
    alist))
```

The following functions check whether we are currently looking at a special line or a line starting with a closing bracket. It returns a pair consisting of the regular expression that matched and the corresponding offset amount.

⟨*Indentation identify special and closing-bracket lines*⟩≡
```
⟨Indentation identify special lines⟩
⟨Indentation identify closing-bracket lines⟩
(defun aldor-looking-at-special-or-closing ()
  (let* (return-value)
    (setq return-value (aldor-looking-at-special-line))
    (if (not return-value)
        (setq return-value (aldor-looking-at-closing-brace)))
    return-value))
```

Do we look at a line that matches one of `aldor-special-line-offset-alist` at the beginning of the line. If yes return the pair from this list. If no then return nil. We assume that point is at the beginning of the line.

⟨*Indentation identify special lines*⟩≡
```
(defun aldor-looking-at-special-line ()
  (let* (
         (return-value nil)
         (special-line-offsets aldor-special-line-offset-alist)
         )
    (while special-line-offsets
      (if (looking-at (caar special-line-offsets))
          (progn
            (setq return-value (car special-line-offsets))
            (setq special-line-offsets nil))
        (setq special-line-offsets (cdr special-line-offsets))))
    return-value))
```

Do we look at a line that starts with a closing bracket as described in `aldor-offset-pair-alist`. If yes return the corresponding pair of regular expression and (negative) offset amount from this list. If no then return nil. We assume that point is at the beginning of the line.

⟨*Indentation identify closing-bracket lines*⟩≡
```
(defun aldor-looking-at-closing-brace ()
  (let* (
         (return-value nil)
         (p nil)
         (offsets aldor-offset-pair-alist))
    (while offsets
      (setq p (car offsets))
      (if (looking-at (concat "^[ \t]*" (cdar p)))
          (progn
            (setq return-value (cons (cdar p) (- (cdr p))))
            (setq offsets nil))
        (setq offsets (cdr offsets))))
    return-value))
```

Move forward one character, but consider an underscore as a escape character and thus move forward two characters in this case.

⟨*Indentation forward-char skipping Aldor escape character*⟩≡
```
(defun aldor-forward-char ()
  (if (looking-at "_") (forward-char))
  (if (not (eolp))     (forward-char)))
```

Move point forward at least one character. Skip strings. Be aware that _ (underscore) is an escape character.

⟨*Indentation move the point behind the string*⟩≡
```
(defun aldor-ignore-string ()
  (if (looking-at "\"")
      (progn
        (forward-char)
        (while (and (not (eolp)) (not (looking-at "\"")))
          (aldor-forward-char)))))
```

3.5.3 Indentation Computation

The indentation algorithm is shortly described at the beginning of Section 3.5.

Probably the most important functions here are the computation of the indentation for the next line (`aldor-compute-next-indentation`) and the function `aldor-indent-line` which does the actual indentation of the line.

⟨*Indentation computation*⟩≡
  ⟨*Compute indentation for the next line*⟩
  ⟨*skip special lines backward*⟩
  ⟨*Indent line*⟩
  ⟨*Indent a line after pressing RET*⟩

Computation of the indentation for the next line is done as follows. We determine the current indentation. Then we adjust the indentation for lines that start with a closing bracket. Finally we scan through the line and adjust the indentation each time we find an open or closing bracket. The adjusting step for lines beginning with a closing is necessary. Since the closing bracket will decrease the offset during the following scanning step, we simply add the corresponding offset amount beforehand.

We assume that we look at the beginning of the line.

⟨*Compute indentation for the next line*⟩≡

```
(defun aldor-compute-next-indentation ()
  (let* (indent offsets forward)
    ; need to adjust the current indentation if the line starts with a
    ; closing brace.
    (setq indent (current-indentation))
    (setq offsets (aldor-looking-at-closing-brace))
    (if offsets (setq indent (- indent (cdr offsets))))

    (if (not (aldor-looking-at-special-line)) ;we might be at line 0
        ;; scan the line and adjust indent
        (progn
          (while (not (eolp))
            (aldor-ignore-string)
            (setq offsets aldor-offset-alist)
            (setq forward nil)
            (while offsets
              (if (looking-at (caar offsets))
                  (progn
                    (setq forward (- (length (match-data)) 1))
                    (setq indent (+ indent (cdar offsets)))
                    (setq offsets nil))
                (setq offsets (cdr offsets))))
            (if forward
                (forward-char forward)
              (aldor-forward-char)))))
    indent))
```

Special lines are considered like empty lines. They do not give any information on the indentation. So we skip backwards until we find a non-special line.

⟨*skip special lines backward*⟩≡

```
(defun aldor-skip-special-lines-backward ()
  (forward-line -1)
  (while (and (not (bobp)) (aldor-looking-at-special-line))
    (forward-line -1)))
```

This is the code that implements the indentation steps described at the beginning of Section 3.5.

Note that we do not modify the indentation of the first line.

⟨*Indent line*⟩≡
```
(defun aldor-indent-line ()
  "Indentation for Aldor major mode."
  (let* (indent i)
    (beginning-of-line)
    (if (> (point) 1)
        (progn
          (save-excursion
            (aldor-skip-special-lines-backward)
            (setq indent (aldor-compute-next-indentation)))
          ; We have the basis indentation.

          ; Now it might happen that we need to indent a
          ; special line or a line that starts with a closing
          ; paren (after initial white space).
          (setq i (aldor-looking-at-special-or-closing))
          (if i (setq indent (+ indent (cdr i))))
          (if (> 0 indent) (setq indent 0))
          (indent-line-to indent)))))
```

After pressing `RET` the cursor will be on an empty line. Thus the indentation would be 0. We therefore go just back to the last non-special line compute the indent and take it without further modification.

⟨*Indent a line after pressing RET*⟩≡
```
(defun aldor-indent-new-line ()
  "Indentation after pressing RET for Aldor major mode."
  (let* (indent i)
    (save-excursion
      (aldor-skip-special-lines-backward)
      (setq indent (aldor-compute-next-indentation)))
    (if (> 0 indent) (setq indent 0))
    (indent-line-to indent)))
```

3.6 Aldor Mode Setup

We set up `aldor-mode`. Note that we do not turn on the syntax highlighting immediately, but it can be done by using `aldor-mode-hook` in your Emacs init file, see Section 2.1.

⟨*aldor-mode-setup*⟩≡

```
;; hook to be run for Aldor mode
(defvar aldor-mode-hook nil)

;; start Aldor mode
(defun aldor-mode ()
  "This is a mode intended to support program development in Aldor."
  (interactive)
  (kill-all-local-variables)
  ;; define own keys
  (use-local-map aldor-mode-map)
  (set-syntax-table aldor-mode-syntax-table)
  ;; indentation
  (set (make-local-variable 'indent-line-function) 'aldor-indent-line)
  ;; font-lock
  (set (make-local-variable 'font-lock-defaults)
       '(aldor-font-lock-keywords))
  ;; set names and run hooks
  (setq major-mode 'aldor-mode mode-name "Aldor")
  (run-hooks 'aldor-mode-hook)
)
```