J. Functional Programming **8** (5): 493–502, September 1998. Printed in the United Kingdom

Dynamic tabbing for automatic indentation with the layout rule

GUY LAPALME

Département d'informatique et de recherche opérationnelle,
Université de Montréal, C.P. 6128, Succ. Centre-ville,
Montréal, Québec Canada H3C 3J7
(*e-mail:* lapalme@iro.umontreal.ca)

Abstract

We show the design principles of an automatic indentation GNU Emacs mode for Haskell and Miranda(tm), functional languages using the 'layout rule' instead of the usual parenthetic structures for indentifying dependent program parts.

Capsule Review

The importance of programming tools is often overlooked; we like to read and write about fancy type systems or clever implementation tricks, but when it comes down to actual programming we all rely on tools like editors and compilers. This article describes a Haskell (and Miranda(tm)) indentation mode for Emacs. While this may not sound so exciting, it is more difficult than one would first suspect because of the layout rule. The indentation mode described here works incredibly well, and once you have used it there is no going back. Use and enjoy!

1 Introduction

A program is a structured text description of the steps involved to solve a problem. Since it is primarily intended for the computer, its physical presentation is not really important because it does not change the meaning of the program. But a program is also a means of formally describing to other humans the steps of solutions and their relations. In this case, physical representation or layout becomes important because it is an intuitive way of making explicit the main steps, their subdivisions and their interrelations.

These steps are often grouped using pairs of keywords such as `begin ... end` or pairs of symbols such as `( )`, `[ ]` or `{ }`. As a program becomes more and more complex, the number of structure levels increases, and it becomes difficult to keep track of them. An obvious way is to indent subparts according to their levels of subdivision by spacing using blanks and/or tabs at the start of each line, a part at a given level being more indented than the part upon which it depends.

As the syntax of computer languages already indicates the part-subpart relations

by means of keywords and special characters, these spacings are not significant for the computer. These relations are thus indicated twice: once for the human and once for the computer.

A grouping like `begin ... end` or `{ }` does not have a distinct semantics and it does not translate into instructions. But as these signs are among the most often used, some propositions have been made to eliminate them altogether and use indentation instead to indicate the dependencies between parts of the program.

Landin (1967) was the first to suggest this, but curiously his suggestion was not followed by language designers of the early 1970s; perhaps this had to do with the fact that programs at this time were often punched onto cards, and thus it was difficult to see the indentation or make corrections to it. Even languages designed more recently (e.g. C++ or Java) still use keywords for indicating part-subpart relations.

This idea was only implemented more than ten years later by David Turner in SASL (Turner, 1979, 1983) and Miranda[1] (Turner, 1985; Research Software Ltd., 1989), two functional languages based on recursive equations. Occam (Inmos, 1984) also used indentation for indicating dependency relations between parts and subparts. In 1990, Haskell language designers also adopted this convention – see Peterson and Hammond (Peterson and Hammond, 1997) for the current version (1.4) of the report. In this paper, we point out the differences between Miranda and Haskell layout rules, but the principles stay basically the same.

We have designed and implemented GNU Emacs modes for both Miranda and Haskell. The algorithms are essentially the same but this paper illustrates the principles using only Haskell. We point out some small differences in layout rules between these two languages but for most cases, the resulting indentation is similar.

2 The 'layout rule'

The principle of the layout rule is deceptively simple, in fact so simple and intuitive that one often uses it without being aware of it even in languages where it is not required. It simply means that a part of a program should be more indented than the part it depends upon.

The Miranda language manual (Research Software Ltd., 1989) defines it as follows:

> Syntactic objects obey Landin's offside rule. This requires that every token of the object lie directly below or to the right of its first token. A token which breaks this rule is said to be 'offside' with respect to that object.

In Miranda, the layout rules applies to any expression or definition but only after certain keywords in Haskell. The Haskell report (Peterson and Hammond, 1997) gives the following definition:

> The layout (or "off-side") rule takes effect whenever the open brace is omitted after the keyword `where`, `let`, `do`, or `of`. When this happens, the indentation of the next lexeme

[1] *Miranda* is a registered trademark of Research Software Inc.

(whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments). For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace. Within these explicit open braces, no layout processing is performed for constructs outside the braces, even if a line is indented to the left of an earlier implicit open brace.

This is a classic example of a simple intuitive idea put in 'legal' terms... but this specification in terms of 'omission of braces and semicolons' ensures a smooth integration of parts of programs that use layout and others that do not.

This slight difference between Miranda and Haskell can result in different interpretations in a few cases like:

```
f x = 1 + x
g y = 1 + y
```

which is interpreted as two functions definitions in Miranda but is taken as:

```
f x = 1 + x g y = 1 + y
```

in Haskell, which is syntactically incorrect. This kind of indentation is a bit devious and most cases pose no special problems.

In both Haskell and Miranda, programs are written by means of recursive equations having three parts:

Left-hand side of an equation giving the name of the function and its parameters, some of which can be constants or patterns that are matched on the input parameters given at the call site;

Guard indicating more complex relations to select an equation for a function;

Right-hand side giving an expression to compute the value of the function should that part of the function be selected for execution.

In Haskell, the left-hand side is separated from the guard by a vertical bar (`|`); an equal sign (`=`) separates the guard from the right-hand side. In Miranda, the guard appears after the right-hand side and starts with a comma.

Figure 1 gives a simple Haskell program to illustrate these points. It is neither the shortest nor the most efficient for this task but it makes the main cases of indentation stand out. The use of layout gives an appearance similar to conventional mathematical notation which is free of syntactic clutter such as `begin ... end` or `{ }`.

A line starting with two dashes (`--`) is a comment. Two colons (`::`) separates the name of the function from its type, so `bdigits` is a function taking an integer as an input parameter and returning a list of integers. The definition of `bvalue` uses an auxiliary internal function `bval` whose definition also obeys the layout rule and is indented accordingly.

```
-- compute the list of binary digits corresponding to an integer
-- Note: the least significant bit is the first element of the list
bdigits               :: Int -> [Int]
bdigits 0             = [0]
bdigits 1             = [1]
bdigits n | n>1       = n `mod` 2 :
                        bdigits (n `div` 2)
          | otherwise = error "bdigits of a negative number"

--  compute the value of an integer given its list of binary digits
--  Note: the least significant bit is the first element of the list
bvalue :: [Int]->Int
bvalue [] = error "bvalue of []"
bvalue s  = bval 1 s
    where
    bval e []                     = 0
    bval e (b:bs) | b==0 || b==1 = b*e + bval (2*e) bs
                  | otherwise    = error "illegal digit"
```

Fig. 1. Examples of Haskell definitions.

3 Indenting the program

The simplest way of indenting is writing the appropriate number of blanks and tabs at the start of each line of the program text, but this can be tedious and error prone, especially in the case of modifications to the structure of the program like adding or removing levels of indentation.

Another way is using a batch program that takes a program as input and produces a 'pretty-printed' version; examples of such programs under Unix are 'vgrind', 'pindent' and 'cb'. But these programs are not very useful during the development of the program, and usually they cannot take care of all problems; comments are not often dealt with satisfactorily.

The easiest way to indent a program is during editing. The program editor can keep track of the levels and indent correctly the next line when one line is terminated. As the indentation depends upon the grammar of the language, one needs a different set of rules for each language. A very popular text editor is GNU Emacs (Stallman, 1984; Halme and Heinänen, 1988), which can be customized using a Lisp dialect in which one can implement a language dependent mode for automatic indentation. Special language modes already exist for most popular languages such as Lisp, C, Pascal, Fortran and Ada, but none of these use layout rules for delimiting statement boundaries. There are already implementations of Emacs editing modes for Haskell (Chitil, 1997), but they use only a very simple minded approach to indenting: they almost always use the same indentation as the one on the previous line. In this paper, we describe the principles we used in implementing an Emacs mode that takes care of the layout rule; we show how this is fundamentaly different from other modes, how it can be implemented, and what are the consequences on the other operations that are usually associated with the editor.

4 Automatic indentation

Within Emacs, indentation is done with the `indent-line` function called by pressing the `TAB` key at the beginning of a line; if the line is empty, the appropriate number of blanks are inserted to indicate the level of indentation and if the line already contains text, that text is reindented if necessary.

From this simple user interface scheme, two other operations are built:

`newline-and-indent` called by pressing `LF` (`Control-J`) is like ending the current line with a `CR` (`Carriage Return`) and pressing `TAB` at the start of the next one. So usually one keys in the program and terminates each line by `LF` ; the program is automaticaly indented as it is entered.

`indent-region` called by pressing `M-C-\` (`\` at the same time as the `Meta` and `Control` keys or pressing `ESC` followed by `Control-\`) is equivalent to pressing `TAB` in front of each line of the 'region' which is a portion of the program text between a 'mark' and the current position of the cursor. By putting the 'mark' and the cursor at the appropriate places in the program, one can thus easily reindent a whole portion of text.

Given this setup, one has only to redefine the `indent-line` function to achieve correct indentation for either a new line of the program or for a region of program text.

4.1 Usual implementation

In current language specific modes, indentation is basically computed as follows: find the deepest currently open structure, i.e. the last open `begin`, `(`, `[`, `if`, `{` without a corresponding `end`, `)`, `]`, `endif`, `}` and indent either directly under or to the right of it.

Emacs Lisp – the language in which the language modes are implemented – already has a primitive function called `parse-partial-sexp` which finds the currently open structures in the case of single character parenthetical structures. This is of great help, but as one often encounters more complicated structures using pairs of keywords, a language-specific mode must not rely solely on this primitive. Furthermore, as this operation is done at the end of each line and that the program is not necessarily syntactically complete, one cannot rely on the usual parsing algorithms. Instead, heuristics and keyword searching are used upstream from the insertion point.

4.2 Implementation with the layout rule

In 'conventional' language modes, a single indentation per line can be determined automatically because indentation is only a matter of presentation that can be inferred from the current program text. However, this is not always the case for languages using the layout rule in which different indentations may change the meaning of the program.

A new line can be indented at various levels according to the intention of the author of the program. Our mode computes a list of all possible 'continuations' of a

line and when TAB is pressed it indents to the rightmost one; should TAB be pressed again, it removes this indentation and indents according to the previous one and so on; when all indentations have been tried, the list is started again. Thus successively pressing TAB rotates between the different indentation points for each line depending on the context; the fact that the user must decide dynamically which indentation point is the correct one is called 'dynamic tabbing'.

As Haskell equations defining the same function but with different patterns start with the same function name, the mode not only computes the indentation, but it also writes the name of the function as one possibility. To start the definition of a new function, one only presses TAB to remove the suggested function name while keeping the same indentation.

Figure 2 indicates how easy it is, to indent while typing the program of figure 1; characters in `this type` are entered by the user. The control characters on the left are interpreted by the Haskell mode; LF ends the current line and indents the next one; TAB goes to the next indentation point and CR ends the current line without indenting. C-cw (Control-C w) inserts the keyword `where` at the appropriate level of indentation. The symbol (_) indicates where the cursor is positioned before the user starts typing; underlined characters are automatically prompted by the system. If the user is not satisfied with the suggested position, he/she only presses TAB to go to the next position; these positions are indicated here on different lines but, on the screen, the cursor stays on the line, but is shifted to the appropriate column; the cursor thus moves to the left, sometimes erasing previously suggested text like the function name. Any other character than TAB ends the indentation cycle and is interpreted in the normal way.

The last four lines of figure 2 illustrate all indenting possibilities for continuing the last statement. A user already knowing that the function is terminated does not have to go through all these cases, but can type CR to go directly to the first column and start the definition of a new function.

In fact, the keys of figure 2 do not give the exact layout of figure 1 in which guards and equal signs are aligned. This alignment is only a matter of keying another command that does this once for the whole function. This alignment also relies on the identification of the three parts of an Haskell statement as described in section 2; it then shifts appropriately all guards and right hand sides and the more indented lines that depend on these parts. This alignment is more easily done incrementally by typing a command that inserts a = or | and then immediately aligns guards and right hand sides from the beginning of the function.

The layout possibilities have been described here in the case of an empty line, but should the line not be empty, it is usually possible to remove spurious indentation possibilities by comparing the start of the line with what is expected at each point of indentation. These restrictions usually insure that the correct indentation is found almost always on the first TAB in the case of non empty lines. This is particularly useful in the case of reindentation of a block of lines using the `indent-region` function described in section 4: the mode goes through each line and prompts the user only for lines for which there are more than one indentations; this usually occurs only at the start of a new function.

```
    ▌ -- compute the list of binary digits corresponding to an integer
CR  ▌ -- Note: the least significant bit is the first element of the list
CR  ▌ bdigits :: Int -> [Int]
LF  ▌
TAB ▌ bdigits 0 = [0]
LF  ▌
TAB ▌ bdigits 1 = [1]
LF  ▌
TAB ▌ bdigits n | n>1 = n `mod` 2 :
LF  ▌                 bdigits (n `div` 2)
TAB ▌
TAB ▌           | otherwise = error "bdigits of a negative number"
LF  ▌
TAB ▌ bdigits
TAB ▌
CR  ▌ -- compute the value of an integer given its list of binary digits
CR  ▌ -- Note: the least significant bit is the first element of the list
CR  ▌ bvalue :: [Int]->Int
LF  ▌
TAB ▌ bvalue [] = error "bvalue of []"
LF  ▌
TAB ▌ bvalue s = bval 1 s
TAB ▌
TAB ▌ bvalue
TAB ▌
C-cw▌     where
LF  ▌     bval e [] = 0
LF  ▌
TAB ▌     bval e (b:bs) | b==0 || b==1 = b*e + bval (2*e) bs
LF  ▌
TAB ▌                 | otherwise = error "illegal digit"
LF  ▌
TAB ▌     bval
TAB ▌
TAB ▌
```

Fig. 2. Possible indentations for a Haskell program.

To get a complete Haskell programming environment within Emacs, we added this indentation scheme to an existing Haskell mode (Moss & Thorn, 1998) which features running an Haskell interpreter in a subprocess, having Emacs parse error messages and pinpoint the location of the error and dynamically completing currently defined Haskell identifiers. It is also possible to use the indentation mode for indentation only. The GNU Emacs-Lisp code for our implementation is available at the following URL: `http://www.iro.umontreal.ca/~lapalme/layout.html`. We now describe how to determine all possible indentation points for a line.

4.3 Determining the indentation points

The first thing to check is for any open bracketed expressions, i.e. between `()`, `[]`, `{}` or within a character string; in this case, the only possible indentation is to the right of the opening symbol.

```
bvalue [] = error "bvalue of []"
bvalue s = bval 1 s
    where
    bval e [] = 0
    bval e (b:bs)
        | b == 0 =
            bval (2*e) bs
        | b == 1 =
            2+bval (2*e) bs
```

Fig. 3. Contour lines for a new line at the bottom are indicated by underlined chars.

Otherwise, we use the following steps:

- find the start of the *current block*,
- determine the *contour lines*,
- create the *indentation points*.

The *current block* is the region before the insertion point in which possible indentation points are searched. Its start is defined as the first character which is not the start of a comment after the nearest line containing only spaces; a line having only a comment is not considered empty, in order to allow comments within definitions. In order to have a consistent indentation for a whole function definition, there should not be empty lines within it, but blank comment lines are accepted. For example, in figure 1, within the definition of `bdigits`, the start of the block is immediately before the `b` in the type definition of `bdigits`. This rule is more efficient if one follows the convention that the top level function definitions are separated by at least a blank line; this is usually done, but the mode does not rely on it.

The *contour lines* are determined by going backward from the insertion point to the start of block and keeping track of the leftmost positions of lines which are not 'hidden' by the following lines. In figure 3, we illustrate this concept on a different layout for the `bvalue` definition given in figure 1. The contour lines for an insertion point at the bottom of figure 3 are given by the underlined characters. Note that most lines of the function are 'hidden', and will not be considered during the computation of possible indentation points. Even though this heuristic gives a partial view of the program, it is not even aware that some lines may be within an `if`, a `case` or a `do` expression, this does not seem to cause problems for indentation. Of course, knowing this, it is possible to create programs which might then be badly indented but, in practice, we never encountered one for which it was a problem.

The *indentation points* are given by the list of column numbers at which to position the cursor for a possible indentation for the current line. This column number can also be associated with a string to insert at this column. The indentation points are computed by considering all possible places where the current line could be aligned with the content of a contour line.

The contour lines are determined from the insertion point and 'pushed' on a list. The top element of the list is then 'popped' to get the starting position of each

contour line to compare with the current line. By going through contour lines from top to bottom and, for each one, from left to right, we 'push' insertion points in an indentation list. In the final indentation list, more indented points will thus appear before the less indented ones. The insertion points can thus be determined in two passes: one from the insertion point to the start of the block to compute the contour line, and one other pass going only through contour lines (only four of the nine in figure 3).

The indentation points for a currently empty line depends upon the column numbers of the three parts of a definition as described in section 2. For example, in a definition having three parts like the following taken from figure 1:

```
bdigits n | n>1       = n `mod` 2 :
```

there are four indentation points:

1. Under the `n` after the `=` sign to continue the right-hand side; in Haskell, it is allowed to continue this part anywhere on the following line but usually it is indented after the `=` like it would be required in Miranda.
2. Under the `|`, to start a new guard for the `bdigit` function, in this case the system already inserts the `|` character.
3. Under `bdigits` to give a new equation for `bdigits`; the system automatically inserts the name of the function
4. Still under `bdigits` but without the name of the function.

If the current line is not empty, then there can be fewer indentation points: for example, if the line starts with a guard, then it can be only aligned with a previous guard or after a standard offset under the name of the function. Analogously, a line starting with an equal sign can only be aligned with a previous right hand side or with a standard offset from the function name. Comment lines are dealt, and are usually indented like the previous line.

For the purpose of this algorithm, a right-hand side starts either with `=`, `::`, `->` or `<-`. So the same algorithm is used to align the type definition of the function with the equal sign or to align alternatives in `case` expression or statements in a `do` expression.

This shallow parsing of a few lines of the program text is quite efficient and does not incur any delay for the user in the editor. However, this simple minded pattern matching can also be fooled by unorthodox indenting rules, for example:

```
  f x
= 1 + x
```

is syntactically correct in Haskell, but the right-hand side will not be considered part of function `f` by our algorithm. Our algorithm will not always find all good indentation points if one mixes explicit braces and semi-colons. But in almost all Haskell programs published (e.g. in the Standard Prelude or in Haskell Libraries), these features are not often used and when they are, they are limited to a single line where indentation rules do not come into play.

5 Conclusion

While the layout rule is a very good rule for the human reading of the program and can be implemented efficiently within the parsing algorithm, the automatic indentation within a program editor raises problems for which we have described a simple and efficient solution. This, coupled with an intuitive user-interface scheme, makes it very convenient to use. It thus cannot be further argued that the layout rule makes it impossible to implement automatic indentation. While fully automatic indentation is not feasible, dynamic tabbing between a few well chosen choices is very convenient and intuitive.

Acknowledgements

We thank Eric Le Saux who first implemented these ideas in a Miranda mode. We also thank Graeme Moss and Tommy Thorn who provided feedback on previous versions of the Haskell indentation mode. We are grateful to anonymous referees who suggested many improvements to the paper.

References

Chitil, O. (1997) *(X)Emacs modes for Haskell.* Available at URL `http://www-i2.informatik.rwth-aachen.de/Forschung/FP/Haskell/`.

Halme, H. and Heinänen, J. (1988) GNU Emacs as a dynamically extensible programming environment. *Software–Practice and Experience*, **18**(10), 999–1009.

Inmos (1984) *Occam Programming Manual.* Prentice-Hall.

Landin, P. J. (1967) The next 700 programming languages. *Comm. ACM*, **9**(3), 157–166.

Moss, G. E. and Thorn, T. (1998) *Haskell editing mode.* Available at `http://www.haskell.org/haskell-mode`

Peterson, J. and Hammond, K. (editors) (1997) *Report on the programming language Haskell, a non-strict purely functional language (Version 1.4).* Department of Computer Science, Technical Report YALEU/DCS/RR-1105, Yale University.

Research Software Ltd. (1989) *Miranda System Manual, version 2.0.*

Stallman, R. M. (1984) Interactive programming environments. In: *EMACS: The extensible, customizable, self-documenting display editor*, Barstow, D. R., Shrobe, H. E. and Sandewall, E. (editors). McGraw-Hill, pp. 300–325.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software – Practice and Experience*, **9**(1), 31–49.

Turner, D. A. (1983) *SASL language manual (revised version).* Technical Report, University of Canterbury at Kent.

Turner, D. A. (1985) Miranda – a non strict functional language with polymorphic types. In: *Conference on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 201*, pp. 1–16. Springer-Verlag.