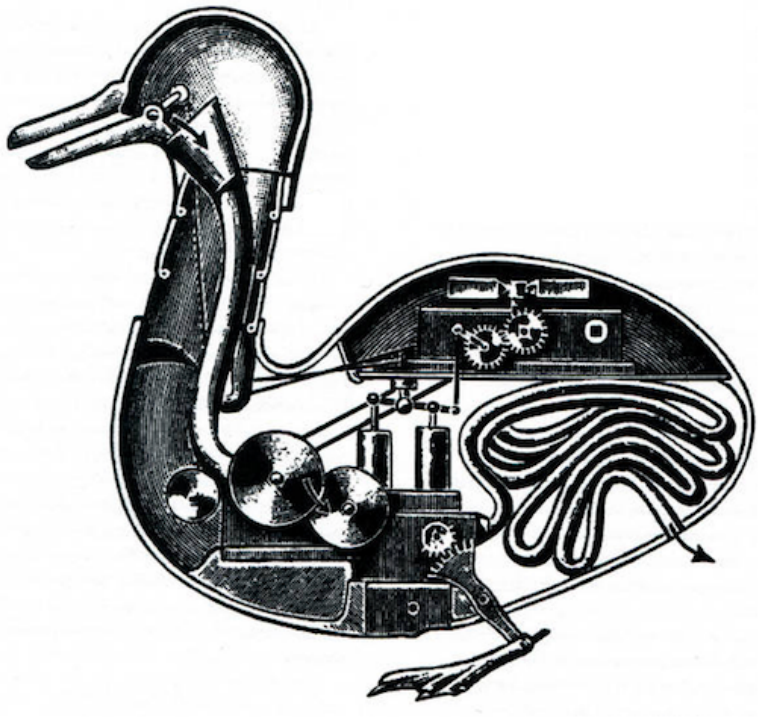
The Patobook

Tom Hirschowitz
Pierre-Etienne Meunier
Christophe Raffalli



Why Patoline ?

Patoline is a new system of typography, aiming at being an alternative to TeX and other word-processing systems. Its name is to be pronounced like “Pa-toe-leen”, and it is the frenchification of the translation in portuguese of a joke in english, as a reflect of the goal of patoline's authors of typesetting any possible language, with any set of symbols, and writing system, possibly in more than one dimension (so as to typeset the language of category theory, for instance). Originally, Patoline was developed without any real name, and tested on a document named “*doc.txp*”. Tom Hirschowitz came with this pun, that we shall call it *Daffy*, so that the name would at least make sense on the test case. Then, Elisa Meunier remarked that our pun made no sense in portuguese, since Daffy Duck's name was different in Brazil. We then decided to change the name, before Pierre Hyvernat and Pierre-Etienne Meunier finally gave it this french touch it has now during one of our project meetings.

This project was started in a french maths laboratory called LAMA, in the french alps. The original idea for a new typesetting algorithm was mine, but would probably have stayed at the state of a mere idea without the support and large contributions of Florian Hatat, Tom Hirschowitz, Pierre Hyvernat, Christophe Raffalli and Guillaume Theyssier.

By the way, as of version 1.0 of Patoline, the goal of “typesetting any language with any symbol set” has not been completely achieved, although significant advances have been done in this direction, and this is still one of our main goal. Moreover, we will certainly recognize the arrival of Patoline 2 by its ability to handle multi-dimensional typesetting.

Patoline is based on the idea that an author should focus on the structure of his documents, and let the machine care about their appearance. However, the fact that it is written and extensible in a modern language makes it easy to change the appearance of documents, in the most generic way known — that is, by using Turing machines — when one wants to do so. This does not necessarily means that anyone who is going to use Patoline needs to forge Turing machines. Instead, it means that the possibility offered to anyone to do so in a simple and fast way, will hopefully yield a lot of high quality extensions within a reasonable amount of time, usable by any Turing-machines-reluctant user to achieve outstanding quality in any document very quickly.

In this book, we tried to follow a logical progression when introducing the concepts and ideas of our system, and to provide progressive exercises to help you to get a good understanding of our system in little time. If there is anything you do not understand perfectly well, do not hesitate to tell us: our common address, at the time of this writing, is `mltypography@googlegroups.com`

1 First documents

1.1 Paragraphs and sections

While it is always possible to compile empty documents with Patoline, these may not be the most interesting ones. Patoline does not have any real graphical interface for now, although it includes a quite good emacs mode, partly generated automatically to match the grammar you select. The instructions are given in chapter 7. So, let us assume that you managed to install patoline on your system, to open a text editor on a blank file. Then, you would write your first Patoline document `hello.txp`:

```
Hello, world !
```

To see the result, just write in a terminal the following command, in the same directory as your file:

```
patoline hello.txp
```

The result should be a pdf file, named `hello.pdf`, with one page containing your text. The choice of pdf files as the default output format is only to avoid breaking the habits of TeX users. Patoline can output its result to other formats, and new output drivers can be written quite easily, as we will see in section 6.4. There is also a way to typeset several paragraphs: by writing them with at least one line of blank space between them, like on the following example:

```
This is my first paragraph.

This is my second one.
```

You may have noticed that the paragraphs all start with an indentation (a space at the beginning of the first line), except if they are immediately after the beginning of a section. This follows the typographic rules described for instance in . If you

are unsatisfied with this behavior, you can change it readily by using the following code:

```
\begin{noindent}
This paragraph is not indented.

Neither this one.
\end{noident}

\indent But this one is.
```

As we already stated above, the preferred encoding system to typeset text in Patoline is UTF-8 (see for instance the wikipedia article for more details). This choice is not completely arbitrary, and we will see later how to change it to your favorite encoding. If you want to test this feature right now, you could write for instance the following sentence in portuguese in a blank file in your favorite text editor, and get a pdf file with the expected result:

```
Patolino é um pato de televisão.
```

Now, you may want to add structure to a document. For instance, to get a document with two sections named “Sex of the angels” and “Reproduction of the angels”, respectively, one would write:

```
=> Sex of the angels

=<

=> Reproduction of the angels

=<
```

This simple code handles section numbering and typesetting for you, and registers these sections for later use in a table of contents, for example:

```
=> Table of contents
```

```
\tableOfContents
=<

=> Sex of the angels

=<

=> Reproduction of the angels

=<
```

The problem, when trying to compile this document, is that the section titled "table of contents" itself gets numbered, and included in the table of contents. There are finer options to control this behavior. Specifically, we could have replaced the couple `=>`, `=<` by `->` and `-<`, or `.>` and `.<`, respectively.

Exercise 1.1.1 Try these commands: in the above example, replace:

```
=> Table of contents
\tableOfContents
=<
```

By a version using `->` and `-<`, then `.>` and `.<`, instead of `=>` and `=<`.

As you may have noticed, a new kind of "command" appeared in our last example: "`\\tableOfContents`". Any item in Patoline beginning with `\\` is not typeset as such, but instead calls a command modifying the output. Of course, there is also a way to get a '`\\`' in the output, by writing "`\\\\`". At the end of this book, you will know how to create all kinds of new commands.

1.2 More structure

Sometimes, the global structure of documents is not restricted to sections and paragraphs. Patoline allows you to use in-text structures, such as numbered lists, and cross-references. The first case we will consider here is a structure called *enumerate*, for creating numbered lists, like for instance the following one, with two items:

1. First item
2. Second item

   To get such a list, write the following Patoline code in your document:

```
\begin{enumerate}
\item First item
\item Second item
\end{enumerate}
```

This way of applying a command on a whole part of the document with `\begin{…}` and `\end{…}` is called an "*environment*". Another example is non-numbered lists; this one is called "*itemize*":

- First item
- Second item

Exercise 1.2.2 Open a new file, and create two lists: one numbered, the other one unnumbered. Then compile your file.

It is not hard to see why *itemize* and *enumerate* are two examples of the same idea of numbering structures; the difference is that the numbering system used by *itemize* is somewhat simpler than the one used by *enumerate*. We will see in section 3.3 how to create new numbering systems generalizing these, and even how to create new environments.

Exercise 1.2.3 Environments `itemize` and `enumerate` both define a new command, called `\\item`. There are many other environments in Patoline, not necessarily linked with text structure. Can you test and tell what the following environments are for ?

- `center`
- `raggedLeft`
- `raggedRight`

There is also a more general enumeration environment:

```
\begin{genumerate}(AlphaLower, fun s -> [tT (s^". ")])
\item First item
\item Second item
\end{genumerate}
```

Which produces:

The environment genumerate takes an argument which is an OCaml value: a pair with two arguments:

- AlphaLower which could also be Arabic, AlphaUpper, RomanLower or RomanUpper. This tells what king of numerals you want to use. Try it!
- fun s -> [tT (s^". ")] is a function taking a string s and return Patoline content. This is descibed later in this book. Here, we add a period and a space after the string which contains the numeral.

There is also a syntactic sugar for genumerate and the following produces the same result:

```
\begin{genumerate}{&a.~}
\item First item
\item Second item
\end{genumerate}
```

With this abbreviation, &1 will be replaced by the item number in numerals and you may guess (or try) the effect of &a, &A, &i, &I. Remark: we have to force the final space to be kept using ~ (we could also use \hspace).

1.3 Theorems and definitions

Finally, there is one more structure that we need when writing math articles: definitions and theorems. These are not defined by default in Patoline. In order to load them, we need to tell Patoline that we are going to use a particular format for our document. Document formats may contain lots of additional commands, and Patoline comes with several different formats. For the moment, since we just want to typeset definitions and theorems, it is enough to use the format for writing articles. This format is called FormatArticle, and we can use it in our document by beginning the file with a special command:

```
(* #FORMAT FormatArticle *)
```

Then, you can get a theorem by writing:

```
(* #FORMAT FormatArticle *)
```

```
\begin{theorem}
This theorem is a theorem
\begin{proof}
The proof is trivial, since the theorem is a tautology.
\end{proof}
\end{theorem}
```

This formats also defines environments “`lemma`”, “`proposition`”, “`example`”, “`definition`”, “`corollary`”,“`hypothesis`”. We will explain later how to define new theorem-like environments.

1.4 Counters

We need to introduce another tool linked with structures, that we can use to write documents, called *counters*. We will see in section 3.2 how to manipulate counters in a more precise way. For now, the only thing we need to know is that there are named counters, that get incremented automatically. One of the major interests of counters is to reference automatically parts of the document. For instance, if we give a *label* to a section, we can reference it:

```
=> Section \label("section with label")

=<

We put a label in section \sectref("section with label").
```

This way, if you add a new section to a huge document of yours, or even worse, if you are collaborating with someone, you will have the guarantee that references follow your initial thought, and not simply a particular stage or version of your work. Notice the syntax: intuitively, the labels are not to be typeset in the final documents: they are just hints, or nicknames, we may give to our sections. Hence the syntax: with parentheses and quotes instead of curly brackets, as these last ones, in Patoline syntax, mean “typeset text”.

A more general way to reference counters is to call them by their name. Actually, `\\sectref` is only a shortcut to the more powerful syntax `\\generalRef("_structure")("name")`. We could rewrite the last example like this:

```
=> Section \label("section with label")

=<

We put a label in section \generalRef("_structure")("section with label").
```

The '_' at the beginning of a counter's name means that it is an internal counter, and that Patoline may manipulate it in a particular way. But actually, the underscore is the only difference: you should avoid creating counters with underscores as the first letter, in order to not get unexpected behaviors when Patoline changes their value. But if you want to touch these counters, fine! Patoline is designed in a way that doing so will merely result in mistakes in your table of contents, but nothing worse.

In the following exercise, you'll understand the full generality of the counter system:

Exercise 1.4.4 Given that the item counter in environments *itemize* and *enumerate* is called "`enumerate`", make a reference to an item of our numbered list example using `\\generalRef`.

1.5 Style

Many persons are dissatisfied with the abilities of typesetting systems to change the styles of their texts: it often requires lots of mouse interaction, frequently obfuscated in "menus" and "dialog boxes", or they have too restricted capabilities. The idea in Patoline is to allow the user to change styles easily. Although we are not able to fully explain the execution model beneath this for now, let us give only a few examples:

- Changing the font size, for instance to 2 millimeters, is done by `\size(2.){Blabla}`. What we call the *"font size"*, or the *"em size"*, in typography, was originally the size of the small metal blocks on which the letters were cast. Since the whole alphabet was cast on blocks of the same size, a "font size" could be defined. Nowadays, with computer typography, this definition has more of an indicative value: no letter can normally get out of the "em grid", but this may exceptionally happen.

  We must signal from now that, although the authors of Patoline all have great consideration for the historical or folkloric measurement units (such as those defined in terms of "feet", "yards" or "inches" of someone), the units used in Patoline are those defined by the *Conférence international des poids et mesures*, usually known as the "International System".

- At the time of writing this book, the color system in Patoline is not completely finished. Indeed, this is a quite complicated topic, as color perception depends on the precise screen, printer, ink quality, that one is seeing a document on. Moreover, each eye sees colors differently. Many proprietary systems have been marketed to graphic designers, even though economic interests of the very companies managing these so-called "standards" may compromise the portability and durability of their work.

  Anyway, a few colors have been defined for now, such as `black`, `white`, `blue`, `green`, `red`, `orange`, `purple`, `pink`, `yellow`, and `gray`. Using them on your text is simply a matter of writing `\color(red){This is red}`. You can also get more by mixing: for instance, `\color(mix 0.3 purple pink){Bla}` gives you the color resulting from mixing 30% of purple and 70% of pink. Composing 40% of the resulting color with 60% of yellow can be done by using parentheses: `\color(mix 0.4 (mix 0.3 purple pink) yellow){Bla}`. The command only asks you the proportion x (between 0 and 1) you want of the first color; the other $1-x$ are automatically filled with the other color.
- Fonts are way simpler (or at least they should be), and Patoline was initiated with the idea that anyone willing to control fine points of his typography should be able to do so, while providing a great default result even without finer adjustments.

  The most complicated point with fonts is that the thing typographs call "*glyphs*" do not correspond perfectly to the intuitive idea of "character". The belief in the opposite has been propagated for quite a long time among software developers by Adobe in all versions of its postscript and pdf proprietary formats, and corresponding software and hardware.

  For instance, as you may have noticed by now, when we wrote "fi" in this book, the result was different from the naive version of simply an f followed by an i. This one would look more like fi.

  To change fonts, here are the commands:

  - `\italic{example}` makes your text *italic*. "Italic", in typography, means that glyphs look as if they were written by hand, and it is not the same as "oblique" or "slanted" fonts (which also exist). For instance, the default font for writing maths in Patoline is italic, but not slanted. Some fonts may have their italic versions also slanted, such as Patoline's default font, called *Alegreya*.
  - `\bold{example}` makes your text **bold**. Classically, heavy use of bold fonts is considered bad style, as it tends to distract the reader's eye. According to Bringhurst , bold fonts are a quite recent addition to the tools of typography, and it is very rarely justified.

- `\sc{example}` typesets your text in Petites Capitales. These are generally used to mark sections without perturbing the "color" of the page (color, in typography, means the ink density on the page). They can also serve to typeset acronyms, when full capitals are not required, or distracting.

The complexity of using other font families is that you need to tell Patoline something about the structure of the new family. But don't worry: Patoline has a pretty good library for handling fonts, and you can use virtually any font you like with Patoline, and even define maths grammars using your favorite fonts. We'll see that later.

Exercise 1.5.5 Can you write ***bold italic text*** ?

1.6 Including from external files

Sometimes, a file gets too long to be easily handled and understood by other systems, such as revision control software, or coauthors. In these cases, Patoline offers a mechanism called *"file inclusion"*, that allows you to split your files, and include them transparently. A consequence of this feature is that you can include the same file from different documents, and even include the same file several times in one document, for instance if it contains a picture. It is also possible to compile the external file alone, for instance to test it, or because you are writing proceedings of a conference, for instance. A special macro is provided to do this, called `\\Include`. Say you have two files, `file1.txp` and `file2.txp`. To include the contents of `file2.txp` from `file1.txp`, you would simply write the following line in `file1.txp`:

```
\Include{File2}
```

Remark the upper-case first letter: this comes from the way Patoline detects dependencies between files. Even if the first letter of `file2.txp` is lower-case on the filesystem, it should be included as `File2`.

1.7 First drawings

A last thing we need to talk about, in this introductory chapter, is a first way to add graphics to your text. For now, let us assume that you have produced a png image called "`pato.png`". To include it, use the command `\\includeGraphics("pato.png")`:

Since most raster graphics do not specify an “optimal” size of the pixels they define, our command `includeGraphics` alone is often not enough, and we need to rescale our pictures. For this purpose, you can use the following command:

```
\id(includeGraphics ~scale=0.2 "pato.png")
```

Even if this may look somewhat cryptic for now, you can simply set `~scale=x` with any value of `x` that you like. Note that even if the scale you chose is an integer, a point is required to make Patoline understand that it is really a decimal number. For instance, you must write “`2.`” instead of simply “2”. We will explain in further detail in the sequel what it means exactly. Anyway, this syntax is by no means Patoline's definitive syntax (any suggestion is welcome!).

2 Mathematical formulas

Patoline's mathematical system is based on a syntax quite close to what you would use to speak orally of mathematics, on the phone, for instance. But, since Patoline is a computing program, and not a fellow mathematician, it is much more picky about what it considers "valid" maths. However, unlike your colleagues, it won't complain if you try to customize its understanding of mathematics.

Patoline's pickiness comes from the fact that it first needs to understand your formulas unambiguously, in order to compute the correct spacing between its symbols. Then, it uses tricky numerical algorithms to do its best and optimize the spacing of formulas. Even though the current version of Patoline does not compensate optical illusions, or precisely computes ink density, this is clearly one of our long-term goals.

The grammar of mathematical formulas is based on a technology called *dypgen*, that allows for ambiguous grammars. If your grammar is ambiguous, that is, the same valid expression may have different meanings, then Patoline will not be able to find the right spacings, and will tell you to correct it. Most of the time, adding curly brackets at the right places is enough to satisfy it.

2.1 Typing maths in patoline

Most mathematical formulas, in patoline, are typed between `$` signs. For instance, writing `$x$` yields the following result: x. Writing operations is not much more complicated: $a + b$ is simply written `$a+b$`. Again, like `\\\\` yielded a `\\` sign in the output, `\\$` can be used to get a plain `$` sign.

There are thirteen classes of special symbols in Patoline:

1. Additive operators, like $+$ or $\cup$
2. Multiplicative operators, like $\cdot$, $\times$ or $\cap$
3. Big operators, like $\sum$ or $\int$
4. Prefix operators, like $+$, $-$ or $\vdash$
5. Postfix operators, like !
6. Arrows, like $\rightarrow$, $\Rightarrow$, $\hookrightarrow$ or $\twoheadrightarrow$
7. Logical connectors, like $\wedge$, $\vee$, $\neg$

8. Relations, like $=$, $\in$, or $\equiv$
9. Quantifiers, like $\forall$ or $\exists$
10. Negations, like $\neg$
11. Punctuation, like ..., . or ,
12. Delimiters, like {}, () or ||||

These symbols all have a way to call them in ascii; that is, they can all be called by a normal Patoline command like `\\int` or `\\forall`, with no special or accentuated characters. However, many symbols have a unicode representation, and Patoline also accepts UTF-8 encodings of these. Several symbols have already been defined in Patoline. Chapter ?? of this book is automatically generated to include all symbols from Patoline's default grammar.

First of all, any symbol can be made a normal symbol by surrounding it with curly braces: to get a $\forall$ in the middle of a sentence, we just have to write , or `{\\forall}`.

To get a formula in *display style*, you may just use two dollar signs instead of one: for instance, `$$a+b$$` will produce the following result:

$$a + b$$

2.2 Extending the maths grammar

3 Macros and environments

Patoline is written in a language called OCaml. This language has several interesting features making it a good language for this kind of projects: it is functional, and a quite good compiler has been written for it, that does the type-inference, typechecking, and optimization job for us. The idea of "focusing on contents instead of typesetting" is thus respected: you do not even have to take care of performance considerations or "runtime errors" yourself: Patoline and OCaml do most of it for you.

To understand how to write macros to simplify and automate common tasks, you will need to understand a little more about Patoline's internal structure, and how it compiles documents. This is the purpose of section 3.1. Then, we will see how to use this model to write your own macros.

Patoline's interface tries to stick with basic OCaml concepts, so that even readers unfamiliar with OCaml programming can find their way quickly through the api. If you find something too complicated, or if you see a possible simplification, we would be happy to hear about it.

3.1 The eight layers of Patoline

Patoline is a layered system. Each layer is a representation of your document, and at each step of the process, a different module is used. This way, if you are dissatisfied with a module or another, you can replace them, and still benefit from the work done in other parts. For instance, experience suggests that few Patoline users will want to rewrite the font or pdf library, while agreeing on an input language is difficult, and writing parsers is quite easy.

1. The first layer is high-level code. That is, the code we have described since the beginning of this book.
2. This code is then translated to ocaml source code.
3. The ocaml code is compiled and linked against an ocaml library called *"Typography"*, and an *output driver* library, such as the one called *Pdf*, or *SVG*. Patoline relies on ocaml tools to automatically detect the dependencies of your document.

4. Then, the resulting program is executed, generating a document structure, which is actually a tree structure. We may write functions to modify this tree, which is the way most "environments" work.
5. This structure is then converted to an array of "*paragraphs*", a paragraph being itself an array of small rectangular boxes, each containing an elementary graphical element.
6. Then, the paragraphs are broken into lines and pages. We call this process the "optimizing layer". The result is an array of pages, which are themselves lists of lines.
7. These lines pass through an "*output routine*", which convert them to basic graphical elements, the same kind of elements that are contained inside boxes at step ??.
8. Finally, the output routine calls an *output driver* with these graphical elements, which can do anything with them, such as writing them into a pdf file, displaying them on the screen, saving them in ocaml format with `output_value`, or anything else.

Macros are always written in OCaml. They may be written in external files, or inlined in your documents. Before beginning more detailed explanations, you may want to check the documentation of Patoline's main library, called `Typography`:

`http://lama.univ-savoie.fr/~meunier/patoline/Typography.doc`

This library is usable independently of the main compiler, by using ocamlfind, for instance, with package `Typography`.

3.2 Text and maths macros

As you probably understood it, the goal of a macro is to generate an ocaml data structure that will be used in the highest level structures of the executable: that is, inside the document tree. There are several different types of contents that a macro may return. For now, we will mainly focus on the text contents, and raw boxes (the "door" to the inferior layer). Let us define a first macro:

```
\Caml(
let a ()=[]
)
\a
```

This macro does nothing: it returns an empty list of content elements, that will be integrated to the document. Not really interesting, but still a good start: for instance, we may already notice that macros always take arguments in Patoline. If no arguments are necessary, then they are given `()` as their only argument by the high-level parser. This is

to control macro evaluation: for instance, if a macro has side effects, then we surely want to execute the side effects each time we call it. Let us add more contents to our macro:

```
\Caml(
let a ()=[tT "Macro"]
)
\a
```

`tT` is the text constructor, at this level of content. It is defined in module `Typography.Document` of package `Typography`. Its argument is a string, that will be converted to glyph boxes before optimization, according to the current font parameters.

As we already pointed out, any argument passed to a macro between curly braces is considered "text to be typeset", that is, arguments at the same level of content as our 'a' macro. To see that, you could do:

```
\Caml(
let a x=x
)
\a{Macro}
```

Now, we need more knowledge of module `Typography.Document`, in order to understand how to modify content structures. Specifically, we need to know that `tT` is not primitive: it allocates a font cache and stores the string, along with this cache, into the primitive constructor called `T`. Let us just ignore this cache, and see what we can do with this constructor:

```
\Caml(
  let caps x=List.map (fun content->match content with
      T (t,_)->tT (String.capitalize t)
    | _->content
  ) x
)
\caps{bla}
```

The other frequently used constructor is `B`, and we use it to generate *raw boxes*, which are the types used in the next level. The interface of boxes is defined in interface

`Typography.Box`, so you can check the documentation of this module for more details. Usually, this constructor is used to make drawing boxes, as in the following example:

```
\Caml(
let dr ()=
  [bB (fun _->
    [Drawing (drawing [Path (default,[rectangle (0.,0.) (10.,10.)])])]
  )]
)
\dr
```

In this example, we needed a few more constructs. The first one is `Path`, defined in `Typography.OutputCommon`. Actually, a drawing is a collection of graphical elements, among which paths, glyphs, hypertext links and raster images. The whole list is given in the documentation of `Typography.OutputCommon`. Again, all measurement units in Patoline are *metric*. The rectangle we have just drawn is actually a square of 1cm by 1cm.

Now, what we call a path is a list of arrays of Bezier curves. Each array of the list is a connected path. More than one element in the list means that the surface we are drawing has holes. I'll let you experiment on that:

Exercise 3.2.1 Draw a rectangle, with a rectangular hole inside.

Another important type of constructor that one may need is called `C`. This is a constructor containing a function of the environment to a list of other content constructors (or `Cs` again!).

For instance, to typeset the state of a counter, we may use:

```
\Caml(
  let counter c=
    [C (fun env->
      let _,counter=try StrMap.find c env.counters with Not_found -> -1,[0] in
      [tT (String.concat "." (List.map string_of_int counter))]
    )]
)
```

A few explanations may be needed in order to understand this command. What we call a *counter* is a couple `(a,b)`, where `a` is called the counter's *level*, and `b` its *value*. The level is meant to control when the counter is reset: every time the `_structure` counter value stack goes below length `a`, `b`'s top of the stack is popped, and replaced by a `0`. A counter level of `-1` means the counter is never reset.

One more word on counters: they are stacks because this is sometimes needed. Recall for instance environment `enumerate`, that we defined in section 1.2. If one of the items uses a new enumerate or itemize environment, the same counter will be used; this is possible only if `\\begin{enumerate}` pushes a new zero on the counter stack.

Now that you understand counters, what about the following exercise:

Exercise 3.2.2 If you recall section 1.4, the most basic counter is called `"_structure"`. Call our freshly defined `counter` macro on this counter. Then put it inside `=>` and `=<`. How does the counter value change with sections ?

There are many things to say about the environment, but maybe the most useful thing you need to know right now is about *names*. The environment has a record field called `names`, in which all the labels of a document are stored, along with their positions and the state of all counters at the time it appeared. Unless you know what you are doing, this record field should never be accessed directly. Instead, you should call function `Typography.Document.names` on the environment to access it. This is to prevent the activation of caches in the case the initially computed position is changed after compilation. Example:

```
=> First section

\Caml(
  let mark labelType name=
    [Env (fun env->
      { env with names=StrMap.add name
            (env.counters, labelType, Layout.uselessLine)
            (names env) })
    ]

  let remark name=
    [C (fun env->
```

```
      let (counters,_,_)=StrMap.find name (names env) in
      let _,str=StrMap.find "_structure" counters in
      [tT ("At the time of typesetting name "^name^
           ", counter _structure was in state ");
       tT (String.concat "." (List.map string_of_int str))]
    )]
)

\mark("test")("example")

=<

\remark("example")
```

Then, upon invoking patoline on this example, it will need to iterate the optimization algorithm, in order to resolve its position. Since we have placed no marker in the document, but only modified the environment, there is no reason the name should be resolved. We need to place a marker, which can be done by modifying the above example like this:

```
=> First section

\Caml(
  let mark labelType name=
    [Env (fun env->
      { env with names=StrMap.add name
            (env.counters, labelType, Layout.uselessLine)
            (names env) });
     bB (fun _->[User (Label name)])
    ]

  let remark name=
    [C (fun env->
      let (counters,_,_)=StrMap.find name (names env) in
      let _,str=StrMap.find "_structure" counters in
      [tT ("At the time of typesetting name "^name^
```

```
          ", counter _structure was in state ");
       tT (String.concat "." (List.map string_of_int str))]
    )]
)

\mark("test")("example")

=<

\remark("example")
```

Since markers are an information used by the optimizer, we need to talk to it directly through *boxes*. The special kind of boxes used to make markers is called `User`. Read the documentation of module `Typography.Box` to find all possible markers.

Moreover, module `Typography.Document` contains everything needed to create and manipulate counters. Just remember that counter names beginning with a `'_'` are reserved, and you may confuse Patoline if you change them. An example internal behavior using this feature is the reset of all counters up to level `x` when counter `_structure` is changed.

3.3 Defining environments

An environment, that is, a global command acting on a portion of the document, is called with `\\begin{bla}` … `\\end{bla}`. In this section, we will see how to define new ones, as well as a few examples of definitions and general methods that you may find useful.

First of all, the following exercise will allow you to get a good understanding of how environments are compiled to ocaml code:

Exercise 3.3.3 Write the following code to a file called `environments.txp`:

```
\begin{test}

\end{test}
```

Now compile this file with `patoline --ml environments.txp`, and examine the resulting file, called `environments.tml`.

Now that you understand how environments are compiled, we need to tell more about how the document structure is represented in Patoline. A document, internally, is an element of type `Typography.Document.tree`. At the beginning of any document, Patoline creates a module called `D`, containing several fields, among which a reference to a zipper over type `Typography.Document.tree`, in a field called `structure`. The idea of this module is to get a "state monad-like" behavior, especially when including other modules, as we saw in section ??. Patoline compiles external inclusions as ocaml files with extension `.ttml`, containing a single functor taking this `D` module as its arguments, and doing the same as a regular `.tml` file would do. This way, each different inclusion of the same file gets instanciated properly at the correct position in the document, independently of the number of times it appears, and the order on the compilation order we give to the ocaml compiler.

3.3.1 Document zippers

Most beginners in Patoline, not acquainted with functional programming, will probably wonder what a *zipper* is. A zipper is nothing more than a data structure, and an marker on a particular position in it. It is a good way to functionally edit structures such as a tree, since we can do the edition locally, without specifying paths to the node we are editing. At the same time, it is a persistent data structure, meaning that a function can "save" a particular version of the tree for later use; the "current version" may still evolve, the saved one will stay the same, and the two versions will share as much memory as possible.

Now, if you look at the type of document structures, e.g. in module `Typography.Document`, it is:

```
type cxt=tree * (int * tree) list
```

Its first component is a tree, the one we are editing. The second component is the sequence of trees we needed to left aside, when walking from the top to the current tree, along with the integer referencing the new subtree, at each step of the walk.

For instance, imagine a function `touch` changing something on document trees. Then, to do the change, while staying at the same position, we could do:

```
\Caml(
   let _=
```

```
      D.structure:= (touch (fst !D.structure), snd !D.structure)
)
```

A cool thing about zippers is that you can "navigate" through them in both directions, as opposed to a tree, where the only possible direction is from the top down. For instance, the following code goes to the upper level, if it exists, and else does nothing:

```
\Caml(
   let _=D.structure:=
     match !D.structure with
       t0,((i0,Node t1)::s)->
          Node { t1 with children=IntMap.add i0 t0 t1.children }, s
     | x->x
)
```

Several functions are defined in module `Typography.Document` to manipulate document zippers. Have a look at the documentation for this module.

Exercise 3.3.4 Using function `top` in module `Typography.Document`, write a macro outputting the document graph in dot. Then compile it with command `graphviz`.

Here is the general things most environments defined in Patoline do:

1. Create a module called `Env_example`, with two functions, `do_begin_env` and `do_end_env`, taking `()` as their only argument.
2. In function `do_begin_env`, do nothing but push the current zipper path (the second component) on a stack. This will ensure that nested environments behave well. The stack does not need to be common to all environments. Defining it inside your newly created environment is fine.
3. In function `do_end_env`, go to the saved position, do some magic on this subtree, then change `D.structure` to this subtree. This way, whatever you do on a subtree stays inside this subtree, and upon exiting your environment, the position inside the document has not changed.

   One possible way of finding your way back into a saved position is the following:

```
D.structure:=follow (top !D.structure) (List.rev (List.hd !env_stack));
env_stack:=List.tl !env_stack
```

Assuming that `env_stack` is the stack where you saved your zipper. Frequently, `do_begin_env` needs to create a new sub-tree, in order for your modification functions to identify the correct part of the document, on which they must act. In this case, an example `do_begin_env` would look like:

```
let do_begin_env ()=
  D.structure:=newChildAfter (!D.structure) (Node empty);
  env_stack:=(List.map fst (snd !D.structure)) :: !env_stack
```

And a corresponding `do_end_env` would be:

```
let do_end_env ()=
  let a,b=follow (top !D.structure) (List.rev (List.hd !env_stack));
  env_stack:=List.tl !env_stack
  (* Do some magic here with a, resulting in a' *)
  D.structure:=up (a',b)
```

Functions `newChildAfter`, `follow` and `top`, as well as the empty node `empty`, are defined in `Typography.Document`.

3.4 Accessing Patoline syntax from within \\Caml

When writing ocaml code, using Patoline's syntax is sometimes needed. For instance, you may want to draw a text containing maths, or simply draw text without caring about `tT` and list syntax. In this case, you would do:

```
\Caml(
let a ()= <<a>>
)
\a
```

To simply write an “a”. You can also include maths:

```
\Caml(
let a ()= <<math example: $a+b$>>
)
```

```
\a
```

Finally, another similar syntax can be used to create new math commands: math list. This one creates a list of `Typography.Maths.math`. Math macros do not need an argument.

```
\Caml(
let a=<$a$>
)
$$\a$$
```

3.5 Compilation options, formats and drivers

Fortunately enough, many patterns occur with quite high frequency when typesetting documents, and we end up writing only very few functions to customize Patoline for each document. In a normal document such as this book, only the title page, and a few drawings, need to use command `\\Caml`.

Also, the same document could be typeset to several different output formats: PDFs, which is the default in Patoline, does not necessarily fit all needs. For instance, the PDF specification for long-time document archiving is still confidential, and no open-source or free compliance verifier exists. In this case, you might want to use open formats such as SVG. But then, your output gets split between lots of different files, which may not be convenient if you want to send them by mail, for instance.

To handle this diversity of uses, we designed Patoline with different *formats* and *drivers*. A format is a collection of functions and environments that may be used in a document, such as `italic`, `itemize` or `theorem`. This is also where output routines are defined; these are the functions that call all the transformation functions to transform the document at layer 4 into an output suitable for the drivers.

3.5.1 Formats

Writing formats requires time, patience, and experience. Looking at `src/Format/FormatArticle.ml` in the Patoline source tree, for instance, will show you an example of how to use the default format to write new ones. At the time of this writing, only two different output routines have been written: one for all paper-based documents, the other one for slides. Output routines are still a somewhat fragile part

in Patoline, and the api may changed quickly to fit more complex situations than it does now.

To select a format other than the default one when compiling your document, for instance one called “OtherFormat”, you can simply call Patoline with option `--format OtherFormat`. There is a better option though: if your document uses format-specific features, or simply if you always want to compile it with the same format, then you can use compilation pragmas. For instance, to always compile your documents with format “OtherFormat”, just write, in the first line of your document:

```
(* #FORMAT OtherFormat *)
```

3.5.2 Drivers

Patoline's system of output drivers is meant to be easy to use and extend. If you wrote a document, then just invoking Patoline with command-line option `--driver GL`, or `--driver SVG`, for instance, is enough to see it with the corresponding drivers. Of course, you can also add a compilation pragma, if you know you'll always want this document to be compiled with a particular driver:

```
(* #DRIVER SVG *)
```

In the particular case of the SVG driver, it creates a directory named `document` (if your document was named `document.txp`), an html file named `index.html`, and svg files in this directory.

4 Drawings

Patoline's drawing system is based on vector graphic primitives, quite usual for anyone that has already used such systems. There are essentially paths made of Bezier curves (for instance, lines are Bezier curves of degree 1), glyphs, and inclusions of raster images.

All the primitives and constructors for this basic layer are in module `Typography.OutputCommon`. There are many ways to interact with these constructs. One of them is to use them directly, the other one is to use Tom Hirschowitz's *Diagrams* library. We explain both in this chapter. Diagrams should be used directly when one need to actually draw something. The more stable api of `Typography.OutputCommon` should be used essentially when writing other libraries.

In the first subsection, you will learn how to draw diagrams. In the second one, we will show an example of writing a library for using graphviz output with Patoline.

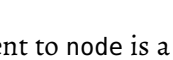
4.1 Using Diagrams

The diagrams library comes in two layers; let's start with the high-level layer.

A first construct to learn is `node`: typing

```
\Caml(open Diagrams)

\diagram(
 let a = node [] <<coin>>
)
```

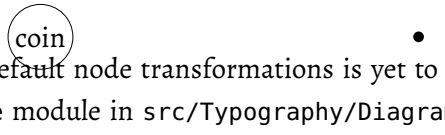
yields



The first argument to `node` is a list of so-called *node transformations*, which are defined in module `Node`. For example, typing

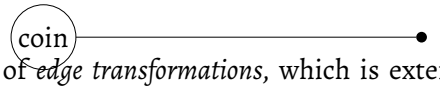
```
\diagram(

 let a = node Node.([circle;draw]) <<coin>>

 let b = node Node.([circle;fill black;at (50.,0.)]) []

)
```

yields

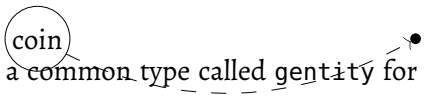


The complete list of default node transformations is yet to be documented; but they are all defined in the `Node` module in `src/Typography/Diagrams.ml`. Node transformations form a directed graph (some should be performed before others), and the library comes with an api to extend it. All existing transformations are actually defined using this api, starting from the empty graph.

A second thing to learn is how to construct edges between nodes. Adding `let e = edge Edge.([draw]) a b` to the previous diagram yields:



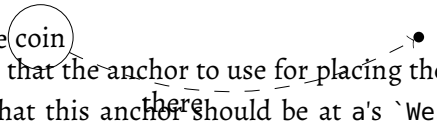
There is also a graph of *edge transformations*, which is extensible using the same api as for node transformations (both are actually obtained by applying the same functor). For example, `let e = edge Edge.([draw;bendRight 30.;arrow env;dashed [2.]]) a b` yields:



Edges and nodes have a common type called `gentity` for graphical entity. Graphical entities have *anchors*, in the Pgf/Tikz sense. E.g., adding

```
 let c = node Node.([at (a.anchor `West);anchor `East]) <<here>>

 let d = node Node.([at (e.anchor (`Temporal 0.3));anchor `North]) <<there>>
```

to our previous diagram gives:



Here, `anchor `East` says that the anchor to use for placing the node `c` is `` `East ``, while `at (a.anchor `West)` says that this anchor should be at `a`'s `` `West `` anchor. Anchors are defined using `OCaml`'s polymorphic variants:

```
type anchor =
  [ `Angle of float (* In degrees *)
  | `North
  | `South
  | `NorthEast
  | `SouthEast
  | `NorthWest
  | `SouthWest
  | `West
  | `East
  | `Center
  | `Main (* The anchor used to draw edges between gentities by default;
             Will be `Center by default. *)
  | `Base
  | `BaseWest
  | `BaseEast
  | `Line
  | `LineWest
  | `LineEast
  | `Vec of Vector.t
  | `Pdf (* The origin when typesetting the contents *)
  | `Curvilinear of float (* Between 0. and 1. (for paths) *)
  | `CurvilinearFromStart of float (* Between 0. and 1. (for paths) *)
  | `Temporal of float (* Between 0. and 1. (for paths) *)
  | `Start
  | `End
  ]
```

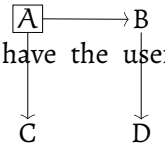
Initially, that was meant to allow extending them with new anchors, but I'm not sure if that's actually possible. So maybe we'll switch to proper variants someday.

For a given graphical entity, not all anchors have to be defined. The only defined node shapes for now are `rectangle` and `circle`. Their `anchor` function signals an error and return the `` `Center `` anchor when they are undefined on their argument.

`Diagrams` comes with a small facility for creating matrices. E.g., typing

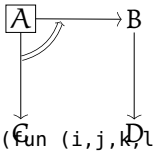
```
  let m,ms = matrix [] [[
  (Node.([draw]),<<$A$>>) ; ([], <<B>>)
];[
  ([],<<C>>) ; ([], <<D>>)
]]

  let edges l = List.map (fun (x,y) -> edge Edge.([draw;arrow env]) x y) l

  let _ = edges (List.map (fun (i,j,k,l) ->
  (ms.(i).(j), ms.(k).(l))) [
  (0,0,0,1) ;
  (0,0,1,0) ;
  (0,1,1,1)
  ])
```

inside a diagram yields



For 2-category inclined people, we have the useful, though still sketchy, refinement of this:

obtained by




```
  let ab :: ac :: _ = edges (List.map (fun (i,j,k,l) ->
  (ms.(i).(j), ms.(k).(l))) [
  (0,0,0,1) ;
  (0,0,1,0) ;
  (0,1,1,1)
```

```
  ])

  let ealpha = edge Edge.([double 0.5 ; bendRight 30.;shorten 0.1 0.3;arrow env;draw])
    Node.(coordinate (ac.anchor (`Temporal 0.3)))
    Node.(coordinate (ab.anchor (`Temporal 0.3)))
```

And that's all for now.

4.2 Using the basic interface

The following code defines a function named `makeGraph`, taking as input options, contents of nodes (in the form of `Typography.Box.drawingBox`), and edges between the nodes, calling graphviz on this graph, and parsing the output to produce a `Typography.Box.drawingBox`.

```
open Typography.Box
open Typography.OutputCommon

let ellipse param x y ah av=
  translate x y
    (Path (param, [Array.map (fun (x,y)->x,Array.map (fun yy->yy*.av/.ah) y)
                      (circle (ah/.2.))]))

let makeGraph opts nodes_ edges=
  let is_space x=x=' ' || x='\n' || x='\t' in

  let inf x=if x= -.infinity || x=infinity then 0. else x in
  let to_inch x=if x=infinity || x= -.infinity then 0. else (x/.25.4) in
  let nodes=Array.map
    (fun (x,y)->
      let cont=(x.drawing_contents x.drawing_nominal_width) in
      let pad=match y with
          `Rectangle->2.
        | `Ellipse->1.
        | _->1.
```

```
    in
    let (a,b,c,d)=bounding_box cont in
    { x with
      drawing_min_width=inf (c-.a)+.2.*.pad;
      drawing_nominal_width=inf (c-.a)+.2.*.pad;
      drawing_max_width=inf (c-.a)+.2.*.pad;
      drawing_y0=inf b-.pad;
      drawing_y1=inf d+.pad;
      drawing_contents=(fun _->
        List.map (translate (pad-.inf a) 0.) cont
      )
    },y
  ) nodes_
in
let i,o=Unix.open_process "dot -Tplain" in
if opts<>"" then
  Printf.fprintf o "digraph {\ngraph %s;\n" opts
else
  Printf.fprintf o "digraph {\n";
Array.iteri (fun i (x,y)->
  match y with
      `Rectangle->
        Printf.fprintf o "n%d [fixedsize=true, width=%f, height=%f,
                              shape=box, label=\"\"];\n"
          i
          (to_inch x.drawing_nominal_width)
          (to_inch (x.drawing_y1-.x.drawing_y0))
    | `Ellipse->
      Printf.fprintf o "n%d [fixedsize=true, width=%f, height=%f,
                            shape=ellipse, label=\"\"];\n"
        i
        (sqrt 2. *. to_inch x.drawing_nominal_width)
        (sqrt 2. *. to_inch (x.drawing_y1-.x.drawing_y0))
    | _->(
      Printf.fprintf o "n%d [fixedsize=true, width=%f, height=%f,
                            shape=box, label=\"\"];\n"
```

```
        i
        (to_inch x.drawing_nominal_width)
        (to_inch (x.drawing_y1-.x.drawing_y0))
    )
) nodes;
List.iter (fun (a,b)->
  Printf.fprintf o "n%d -> n%d[arrowhead=none];\n" a b
) edges;
Printf.fprintf o "}\n";
close_out o;




let rec next_token (buf,pos,f)=
  (if !pos >= String.length !buf then (buf:=input_line f;pos:=0));
  let pos0= !pos in
  while !pos<String.length !buf && not (is_space !buf.[!pos]) do
    incr pos
  done;
  let s=String.sub !buf pos0 (!pos-pos0) in
  incr pos;
  if s<>"" then s else next_token (buf,pos,f)
in
let skip_line (a,b,c)=a:=""; b:=0 in
let number s=float_of_string (if String.contains s '.' then s else s^".") in
let of_inch x=(number x)*.25.4 in
let buf=(ref ""), (ref 0), i in
let w=ref 0. in
let h=ref 0. in
let rec parse l=
  let t=try next_token buf with _->"" in
  match t with
      "graph"->(
        let scale=number (next_token buf) in
        w:=scale*.(of_inch (next_token buf));
```

```
    h:=scale*.(of_inch (next_token buf));
    parse l
  )
| "node"->(
  let name=next_token buf in
  let i=int_of_string (String.sub name 1 (String.length name-1)) in
  let x0=of_inch (next_token buf) in
  let y0=of_inch (next_token buf) in
  let x=x0-.((fst nodes.(i)).drawing_nominal_width/.2.) in
  let y=y0-.((fst nodes.(i)).drawing_y1+.
              (fst nodes.(i)).drawing_y0)/.2. in
  let w=of_inch (next_token buf) in
  let h=of_inch (next_token buf) in
  skip_line buf;
  parse (
    (match snd nodes.(i) with
        `Rectangle->
          [Path ({default with lineWidth=0.1; close=true},
                 [rectangle
                     (x,y+.(fst nodes.(i)).drawing_y0)
                     (x+.(fst nodes.(i)).drawing_nominal_width,
                      y+.(fst nodes.(i)).drawing_y1)])]
      | `Ellipse->
        [ellipse
            {default with lineWidth=0.1; close=true} x0 y0 w h]
      | _->[]
    )
    @(let node,_=nodes.(i) in
      List.map (translate x y)
          (node.drawing_contents node.drawing_nominal_width))
    @l
  )
)
| "edge"->(
  let _=next_token buf in
  let _=next_token buf in
```

```
      let n=int_of_string (next_token buf) in
      let x0=of_inch (next_token buf) in
      let y0=of_inch (next_token buf) in
      (* Dot output is a sequence of splines. We need to parse this
         to get a path with plain Bezier curves. *)
      let rec spline n x1 y1 l=
        if n<=0 then (List.rev l) else (
          let x=Array.make 4 x1 in
          let y=Array.make 4 y1 in
          for i=1 to 3 do
            x.(i)<-of_inch (next_token buf);
            y.(i)<-of_inch (next_token buf)
          done;
          spline (n-3) x.(3) y.(3) ((x,y)::l)
        )
      in
      let path=spline (n-1) x0 y0 [] in
      skip_line buf;
      parse (Path ({default with lineWidth=0.1}, [Array.of_list path])::l)
    )
    | ""
    | "stop"->
      {
        drawing_min_width= !w;
        drawing_nominal_width= !w;
        drawing_max_width= !w;
        drawing_y0=0.;
        drawing_y1= !h;
        drawing_badness=(fun _->0.);
        drawing_contents=(fun _->l)
      }
    | x->(Printf.fprintf stderr "Parse error on input %S" x;exit 1)
in
parse []
```

The first part of this function starts dot, then outputs the graph in dot syntax to its standard input. The standard output is then read by function `next_token`, and converted to constructors of `OutputCommon.raw`.

Since the result of `makeGraph` is a `drawingBox`, and we need contents lists in the document tree, the way to use this function is the following:

```
\Caml(
  let graph=[bB (fun _ ->
    let a=…
    and b=…
    and c=… in
    [Drawing (makeGraph a b c)]
  )
)
```

For instance, to create a graph with two nodes and an edge between them, you would do:

```
\Caml(
  let graph=[bB (fun env ->
    let opts="[ranksep=0.15,nodesep=0.15]"
    and nodes=[| drawing (Document.draw env <<A>>);
                 drawing (Document.draw env <<B>>) |]
    and edges=[|(0,1)|] in
    [Drawing (makeGraph nodes edges)]
  )]
)
```

If you look at the types, you'll see that `Document.draw` outputs raw drawing elements (of type `Typography.OutputCommon.raw`), whereas `makeGraph` needs “drawing boxes”. A drawing box is nothing more than raw graphic elements with a bounding box around them. `draw` is a function computing boundaries, and making boxes out of raw graphic primitives.

One thing you get for free when drawing in Patoline is compatibility with all the drivers in Patoline. For instance, this code produces an output usable by the SVG driver on a web page, or by the OpenGL driver in a presentation.

5 Understanding the typesetting model

This chapter describes the typesetting model used by Patoline. Few users will actually need this part, unless they are trying to write complicated environments such as `itemize`.

5.1 How it works

Patoline's optimizer is yet another example of dynamic programming, on a somewhat more complex space than TeX was. You may want to have a look at Knuth's article describing the algorithm in TeX in . The idea is always the same: cut a document into two parts, prove that you can typeset them independently, then typeset them and concatenate compatible results.

The idea in Patoline is that a *position* in a document is given by a variety of parameters: the semantic position in the text (the index of a box in a paragraph), the number of figures already placed, the current page number, and the vertical height on this page. Instead of being boxes, the badnesses are *functions*, with the ability to evaluate some aspects in the global document. For instance, at each point, a badness function can know the current page, the position on that page, and the number of figures already placed, along with their position.

So, the idea is to build a graph, where each vertex is either a line of text or a figure, placed on a page of the document. The edges all have a distance, which is the badness of moving from a line to the next one. Once the graph is built, Patoline's optimizer uses Dijkstra's algorithm to find the shortest path between the first and the last line.

The algorithm has a notion of “current nodes”. For each of them:

1. A list of all the possible continuations is computed.

5.2 Controlling the optimizer

One thing to keep in mind, when writing such functions, is that they may be called by the optimizer several times, on different attempts to typeset the document. But only one of these attempts will be selected. So write all the functional parts of your functions as if

the document was the final version, but be careful if you use imperative features of ocaml, such as references, or other kind of side effects.

5.2.1 Parameters functions

5.2.2 Completion and badness

6 Extending Patoline

7 Patoline and text editors

When you compiled Patoline, it generated a free emacs mode in a directory called `emacs`, at the root of the Patoline source tree. In order to use it, copy all the files in this directory to some place on your file system, say `/path/to/patoline`, and then append the following lines to your `~/.emacs`:

```
(add-to-list 'load-path "/path/to/patoline/")
(require 'patoline-mode)
```

This mode also gets installed to a default location emacs knows of, when you install Patoline by invoking `make install` at the root of the Patoline source tree. In order to use it, you must install `mmm-mode` and `tuareg-mode`.

8 License

Patoline itself is distributed under the terms of the Gnu General Public License.

9 Bibliography