Haskell√LaTeX

A Haskell pretty-printer

Candidate number: 13243

Computation undergraduate project
Trinity Term, 2002

Haskell λ LaTeX

A Haskell pretty-printer

Candidate number: 13243

Computation undergraduate project
Trinity Term, 2002

Summary

Haskell[1] allows a script to be written in a literate style. It is designed such that a script written in this style can be written in a TeX style with the executable code surrounded by `\begin{code}` and `\end{code}`.

While neither plain TeX nor LaTeX have a `code` environment it is trivial to define a simple environment to show the code. For example, in LaTeX, one can make it an alias for the verbatim package with

```
\usepackage{verbatim}
\newenvironment{code}{\verbatim}{\endverbatim}
```

However, researchers publishing documents containing Haskell code do not tend to print the code verbatim, but prefer to make use of the typesetting features available such as printing keywords in **bold**, `->` as $\rightarrow$ and the type variable `a` as α. At the time this project was started there were three tools for typesetting Haskell in TeX or LaTeX available [6,8,14] and towards the completion of this project a fourth was released [31]. However, as we will see, all four have their shortcomings.

Buried deep in the GHC source code we find the following words of wisdom:

> It is hoped that this style of programming will encourage the writing of accurate and clearly documented programs in which the writer may include motivating arguments, examples and explanations.
>
> — fptools/ghc/utils/unlit/unlit.c
> in the Haskell CVS repository [13]

However, few, if any, such scripts exist. Perhaps part of the reason is the difficulty and lack of automation in turning such a script into a nicely typeset document. It is the aim of this project to produce a Haskell program which, when run on a literate Haskell script, will produce a LaTeX document that is the same as the original but with the code sections typeset nicely.

[1] Throughout this report the term "Haskell" will be used to refer to Haskell 98 [27].

Contents

List of Figures

1 Introduction

1.1 Literate programming

Knuth [16] is generally regarded as the founding father of literate programming [25]; his paper with this name [20] in 1984 introduced the concept to the world and his book of the same name [22] in 1992 expands on the ideas. In his original paper he tells us:

> I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."
>
> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Among the literate programming systems that exist today is Knuth's `WEB` system [19], introduced by his paper, in which a literate script is composed of multiple sections, each of which can have TeX documentation, macro definitions and Pascal code. The utility `WEAVE` can then be used to produce a TeX document comprised of the TeX documentation in the script along with the macros and Pascal code pretty-printed, while its companion tool `TANGLE`, when given the same WEB script, takes the code part of each section and arranges the pieces in the correct order to create a correct Pascal program.

Numerous other literate programming systems exist for a large number of programming language and documentation language combinations. The Haskell report defines a style of writing Haskell scripts which provides many, but not all[2], of the features we would expect of a literate programming system. A significant difference to the `WEB` system is that Haskell compilers and interpreters work directly on the literate script.

Once a format in which literate scripts can be written has been defined two things are needed to complete the literate programming system. Firstly it must be possible to take a literate script and either interpret it or compile it to executable code. This by itself would only constitute a conventional programming system; for a literate programming system it must also be possible to take a literate script and produce the document described by the script. Haskell is fortunate enough to have three separate, high quality implementations of the former but is sadly lacking in the latter. We aim here to redress the balance.

[2] Notably it does not allow functions to be written in out of order fragments. This is less important in a language like Haskell where there are no variable declarations and values can be bound to variables out of order.

1.2 This project

We will create a tool which takes a literate script and extracts the code fragments like:

```
\begin{code}
qsort :: [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort xs_lt_x ++ [x] ++ qsort xs_greq_x
    where xs_lt_x   = [y | y <- xs, y < x]
          xs_greq_x = [y | y <- xs, y >= x]
\end{code}
```

These fragments are then parsed and LaTeX markup for the pretty-printed code is substituted, giving a result like this:

$$\begin{array}{l}
\mathit{qsort} :: [\alpha] \rightarrow [\alpha] \\
\mathit{qsort}\ [\,] \qquad = [\,] \\
\mathit{qsort}\ (x{:}\mathit{xs}) = \mathit{qsort}\ \mathit{xs\_lt\_x} +\!\!+ [x] +\!\!+ \mathit{qsort}\ \mathit{xs\_greq\_x} \\
\quad \textbf{where}\ \mathit{xs\_lt\_x} = [y \mid y \leftarrow \mathit{xs},\ y < x] \\
\qquad\qquad \mathit{xs\_greq\_x} = [y \mid y \leftarrow \mathit{xs},\ y \geq x]
\end{array}$$

All of the code in this report, including the *qsort* example immediately above, has been typeset using the program developed.

This report continues with a discussion of relevant existing theory and technologies, including a look at existing Haskell pretty-printers, in Section 2. This is followed by the writeup of the bulk of the work, the parser, in Section 3—this is largely the development of a parser combinator library, described in Section 3.3. The remainder of the work, the output phase, follows in Section 4. We conclude with a discussion of the value of the work and possible future developments in Section 5.

This project makes a number of contributions back to the community. These include a flexible parser combinator library, a parser for Haskell built upon this library and a pretty-printer for Haskell built upon this parser.

2 Existing theory and technologies

2.1 Haskell

Haskell [2] is a polymorphically typed, lazy, purely functional language. The language specification allows a style of script that encourages the typesetting of both code and description in a TEX document. This is both the language which we will pretty-print and the language in which we write the pretty-printer. We choose to write the pretty-printer in Haskell partly due to the strength of the language in parsing and tree manipulation and partly because we believe the basic tools for any self-respecting language should be written in that language.

An introduction to the Haskell language is beyond the scope of this report. Readers unfamiliar with the language are encouraged to read one of the books available on the subject [5, 30].

2.2 Parsing

The art and science of parsing has been of interest to computer scientists for decades and much of the current theory and practices in use today date from the 70s and 80s. Indeed the canonical reference [4] was published in 1986.

The normal overall strategy is to first *lexically analyse*[3] the input to produce a list of tokens from the list of characters, normally dropping the whitespace and comments, and then *syntactically analyse*[4] the list of tokens to get an abstract syntax tree.

The two most common strategies employed for syntax analysis are called top-down and bottom-up. Bottom-up SAs start from the terminal symbols and build up to progressively larger language constructs; typical examples are so-called *shift-reduce parsers* which generally consist primarily of a large table describing a state machine. As such they tend to be generated mechanically from an abstract description of the grammar. Top-down SAs, i.e., starting with the root non-terminal and substituting non-terminals for one of their productions in a depth-first fashion, are much easier to write and check by hand but efficient implementations can only be created for a more restrictive set of grammars.

2.3 Parsing with parser combinators

Parser Combinators, or PCs, provide a way of building lexical analysers and top-down syntax analysers by combining smaller parsers as the name suggests. Typically you will be provided with (at least):

Fail A parser that consumes none of a list of items of type α and fails.

Succeed A function that takes a value x of type β and returns a parser that consumes none of a list of items of type α and successfully returns x.

Match A function that takes a value x of type α and returns a parser that takes a list of items of type α and either successfully returns x and removes the first item of input from the list if the first item in the list is equal to x or fails otherwise.

Sequential Composition A function that, given two parsers, applies the first parser and, if it succeeds, applies the second to the input as it is after the first parser has been applied. If either of the parsers fails then the sequential composition of the two fails. If both are successful then what happens depends somewhat on the implementation. In the Haskell world where we are blessed with functions as first class values it is common for the result to be the value returned by the first parser applied to the value returned by the second parser.

Choice A function that, given two parsers, decides in some way whether to return the result of the first or the second. Both parsers are tried on a copy of the original input stream and the new input stream is as it was after the chosen parser was applied to it. Common rules for determining which to chose are:

[3] *lexically analysing* is commonly shortened to *lexing* and also known as *scanning*.

[4] *syntactical analysing* is also commonly refered to as *parsing*, but we will avoid this usage, instead referring to the process of lexical analysis followed by syntax analysis as parsing.

- If the first parser successfully consumes any input or the second parser fails then return the result of the first parser. Otherwise return the result of the second parser, which could be a failure if both parsers fail.
- If both parsers succeed then return the result of the one that consumes the most input, or the first if they consume the same amount of input as each other. If only one succeeds then the result of that parser is returned, otherwise failure is returned.

Two general purpose implementations of parser combinator libraries for Haskell exist [24, 29] as well as a document explaining how monadic parser combinators can be implemented in a Haskell-like language [15].

2.4 Typesetting in (plain) TEX

Typesetting, "the act or art of setting type" [28], dates back to before programmable computers even existed when printing companies typeset documents by hand to be printed on printing machines. As computers appeared, so too did computer based typesetting systems, although these were hardware and OS specific.

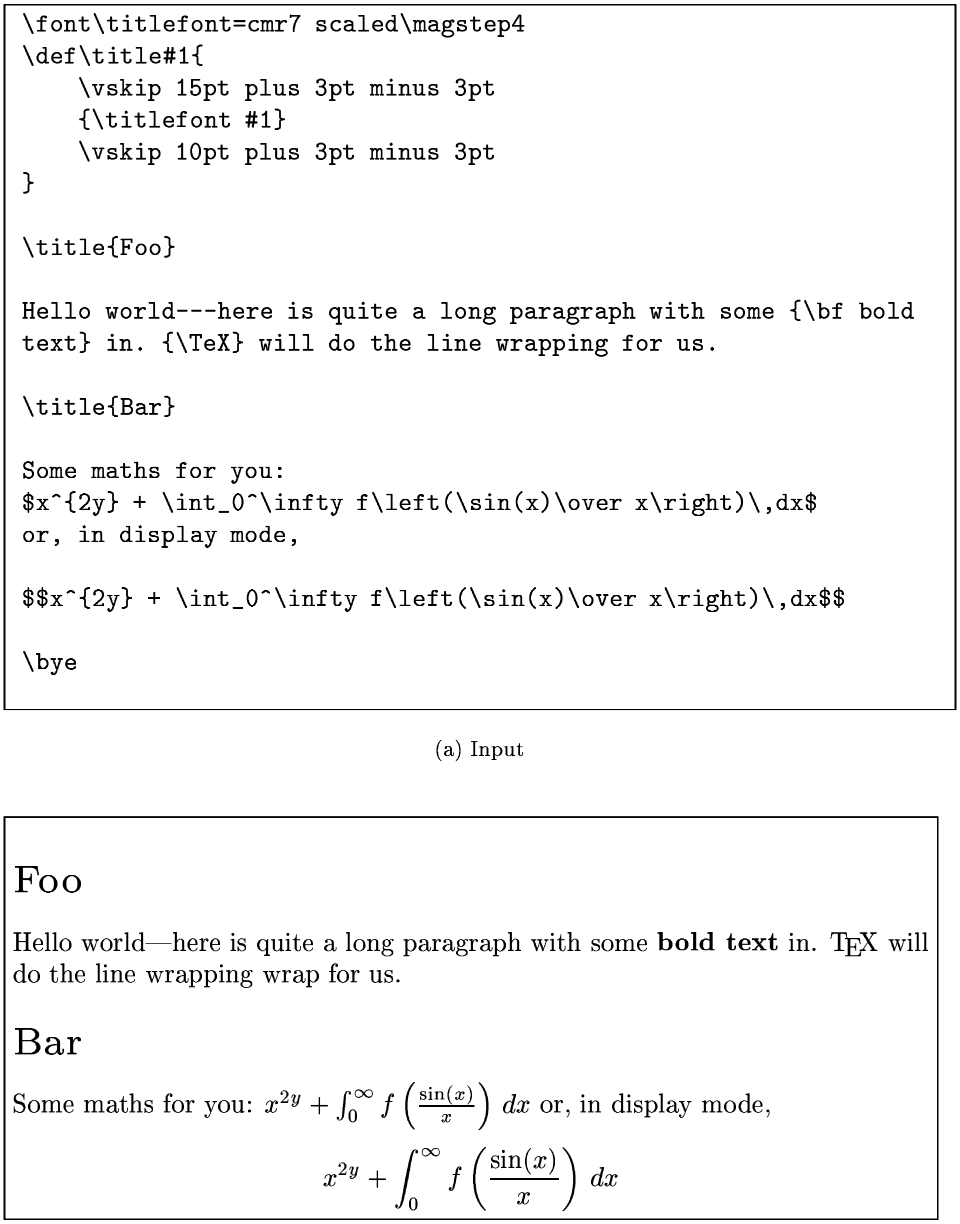
When Knuth saw Addison-Wesley's typesetting of the 2nd edition of volume 2 of "The Art of Computer Programming" he was appalled and so, in May 1977, started work on TEX, which later he described as "a new typesetting system intended for the creation of beautiful books—and especially for books that contain a lot of mathematics" in the preface of the TEXbook [21]. Two versions of TEX exist—TEX78 and TEX82. The former is now considered obsolete and TEX is taken to mean TEX82 unless otherwise specified. A prototype was started in the summer of 1977 followed by a complete implementation from late 1977 to early 1978. Version 0 was released in September 1982, and version 1 in November 1983. The current version, 3.14159, was released in March 1995 and has had only minor corrections to the documentation since—a remarkable stability record compared to other complex pieces of software.

For more information on TEX the definitive reference is the TEXbook [21], although the USENET group comp.text.tex and [9] are also useful. For more on the history of TEX see [17, 18] and try searching for "history of TeX" in comp.text.tex on Google Groups [1]. This report will assume the reader is familiar with the basics of using TEX to typeset documents.

TEX itself has only the primitive commands defined. When people talk about writing in TEX they generally really mean plain TEX—although some of the more confused actually mean LATEX which we will look at in the next section! A simple plain TEX document source and output is shown in Figure 1.

2.5 LATEX

The LATEX companion [12] tells us "LATEX is a generic typesetting system that uses TEX as its formatting engine.". It was written in the early 1980s by Leslie Lamport who released 2.09, his final release, in April 1986. Then maintenance was taken over by the LATEX3 Project [3] who are responsible for LATEX 2_ε, the current release.



```
\font\titlefont=cmr7 scaled\magstep4
\def\title#1{
    \vskip 15pt plus 3pt minus 3pt
    {\titlefont #1}
    \vskip 10pt plus 3pt minus 3pt
}

\title{Foo}

Hello world---here is quite a long paragraph with some {\bf bold
text} in. {\TeX} will do the line wrapping for us.

\title{Bar}

Some maths for you:
$x^{2y} + \int_0^\infty f\left(\sin(x)\over x\right)\,dx$
or, in display mode,

$$x^{2y} + \int_0^\infty f\left(\sin(x)\over x\right)\,dx$$

\bye
```

(a) Input

Foo

Hello world—here is quite a long paragraph with some **bold text** in. TEX will do the line wrapping wrap for us.

Bar

Some maths for you: $x^{2y} + \int_0^\infty f\left(\frac{\sin(x)}{x}\right)\,dx$ or, in display mode,

$$x^{2y} + \int_0^\infty f\left(\frac{\sin(x)}{x}\right)\,dx$$

(b) Output

Figure 1: Example plain TEX document

Among the additional features offered by LaTeX are a package system and many packages; we will find `tabularx`, `ifmtarg` and `verbatim` particularly useful. For further information on LaTeX, see [3,12,23,26]. An example of a simple LaTeX document is given in Figure 2. As you can see, it is a higher level language with a more structured appearance. This report will assume the reader is familiar with the basics of using LaTeX to typeset documents.

2.6 Pretty-printing Haskell in TeX and LaTeX

This is not the first attempt to produce a package that pretty-prints Haskell scripts in TeX or LaTeX. The "Libraries and Tools for Haskell" section of the Haskell webpage [2] has a subsection "Typesetting Haskell in TeX" in which four packages are listed [6,8,14,31], one of which was only released towards the end of this project.

2.6.1 Haskell Style for LaTeX2e

The first [8] is not intended to take a Haskell program and typeset it. Instead it provides a set of commands with which a Haskell program can be typeset by hand. Although these give the author full control over the final layout it means that the code is no longer executable.

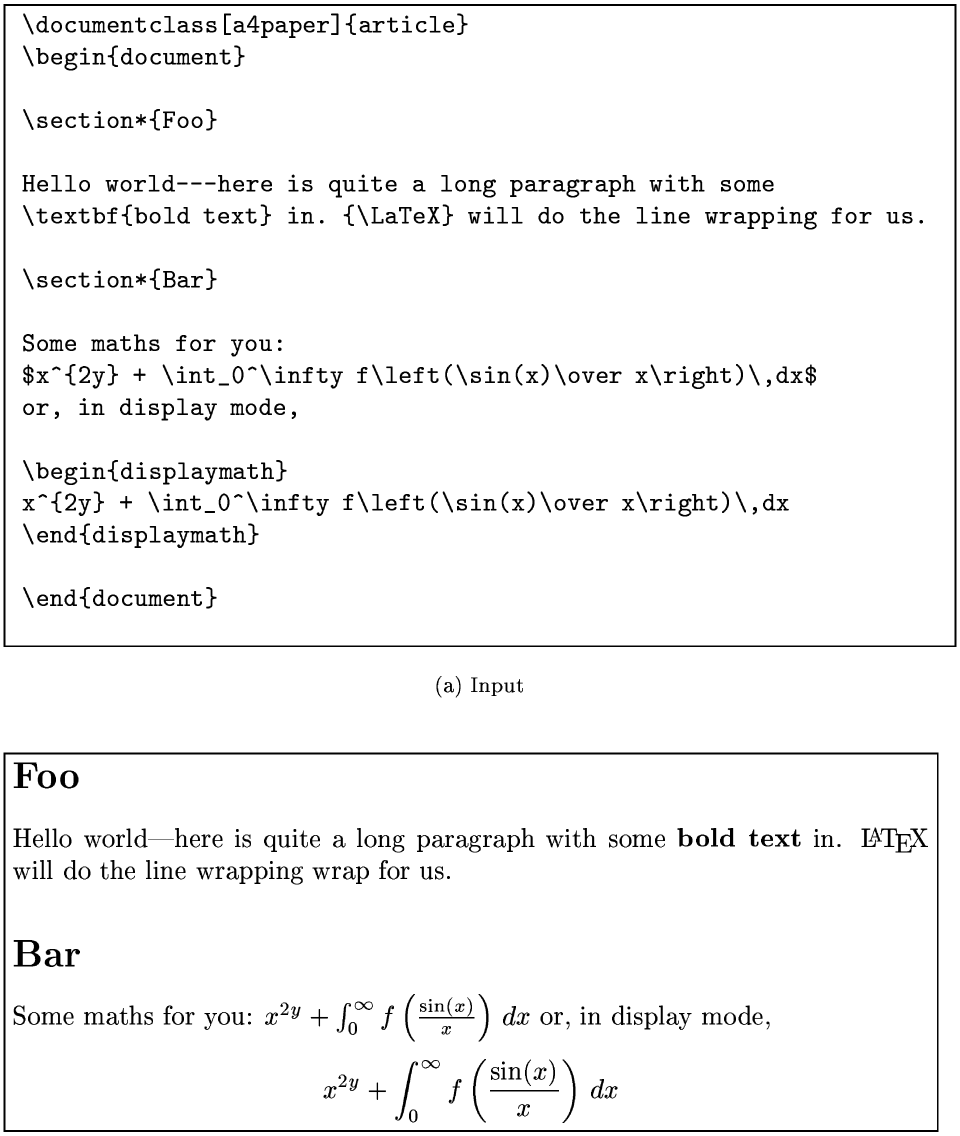
2.6.2 haskell.sty

The next contender, `haskell.sty` [6], is a simple LaTeX style file which uses the listings package to do the pretty-printing. It makes no attempt to print keywords in bold and, while it will correctly pretty-print => as $\Rightarrow$ it is easily confused and will, for example, pretty-print <=> as $\leq>$.

2.6.3 λTeX

The way the author of λTeX [31] wrote his `Example.lhs` shows the package at its best, with the code being written in a similar style to the code below. In particular note that keywords starting blocks are on a line of their own and the first line of the block is on the next line. Also note that each level of indentation is a single space.

```
> foo =
>  do
>   putStrLn "Hello world!"
>   putStrLn "Goodbye!"
> foo = bar1 + bar2
>  where
>   bar1 = 2 * bar1'
>    where
>     bar1' = 3
>   bar2 = 2
```



```
\documentclass[a4paper]{article}
\begin{document}

\section*{Foo}

Hello world---here is quite a long paragraph with some
\textbf{bold text} in. {\LaTeX} will do the line wrapping for us.

\section*{Bar}

Some maths for you:
$x^{2y} + \int_0^\infty f\left(\sin(x)\over x\right)\,dx$
or, in display mode,

\begin{displaymath}
x^{2y} + \int_0^\infty f\left(\sin(x)\over x\right)\,dx
\end{displaymath}

\end{document}
```

(a) Input

Foo

Hello world—here is quite a long paragraph with some **bold text** in. LaTeX will do the line wrapping wrap for us.

Bar

Some maths for you: $x^{2y} + \int_0^\infty f\left(\frac{\sin(x)}{x}\right)\,dx$ or, in display mode,

$$x^{2y} + \int_0^\infty f\left(\frac{\sin(x)}{x}\right)\,dx$$

(b) Output

Figure 2: Example LaTeX document

The typeset version is below. This looks like valid Haskell as intended.

```
foo =
  do
    putStrLn “Hello world!”
    putStrLn “Goodbye!”
foo = bar_1 + bar_2
  where
    bar_1 = 2 ∗ bar'_1
      where
        bar'_1 = 3
    bar_2 = 2
```

However, suppose we now write some code in a style preferred by this author.

```
> foo = bar1 + bar2
>   where bar1 = 2 * bar1'
>             where bar1' = 3
>         bar2 = 2
```

When this is typeset by the package, not only is the second **where** keyword indented far more than we would like, the code no longer looks valid—the definition of *bar2* is indented more than the definition of *bar1*.

```
foo = bar_1 + bar_2
    where bar_1 = 2 ∗ bar'_1
                       where bar'_1 = 3
               bar_2 = 2
```

This is a necessary limitation of any pretty-printer which works only on individual tokens and does not use a monospaced font.

Also conspicuous by their absence in `Example.lhs` are case statements. Both styles of laying out case statements that this author is familiar with are shown below.

```
> foo x = case x of
>             0 -> 0
>             _ -> 1
> foo x = case x of 0 -> 0
>                   _ -> 1
```

As the typeset version below shows, the first of these has the caselets further to the right than is aesthetically pleasing and the second does not have the second caselet aligned with the first, as with the **where** example above. The problem is of course the same limitation as mentioned above.

```
foo x = case x of
                       0 → 0
                       _ → 1
foo x = case x of 0 → 0
                                   _ → 1
```

```
foo         = x + y
    where x = 10
          y = 20
```

Figure 3: Example lhs2TeX input

2.6.4 lhs2tex

Perhaps the best thought out is lhs2TeX [14]. It offers two output modes, one using a monospaced font and the other a proportional font. The monospaced font mode gets around the problems with lexing only pretty-printers, but the result is less pleasing to the eye. In the other mode an alignment column can be specified and everything from that column onwards is aligned; for example, in the example input in Figure 3 you would set the alignment column to 13 so the right hand sides were all aligned. This is an interesting idea, although it would be better if it could be specified more locally.

2.6.5 Related programs

There are a few other programs which do related tasks; we mention them here for completeness. The filter detex removes LaTeX and TeX control sequences; when a literate script is filtered through it reasonable plain text documentation is output. The GNU enscript implementation comes with a Haskell pretty-printing mode which can be used to highlight Haskell scripts with either boldface or colour; unfortunately it gets confused and goes wrong very easily. Syntax highlighting modes for a number of editors exist and some also support automatic indentation. There are also three Haskell source browsers: HDoc, HaskellDoc and “The Haskell Module Browser”; these have varying support for user supplied documentation.

2.6.6 Doing better

As we saw earlier in this section, the two serious packages both run into problems with simply lexing and printing in a proportional font, although lhs2TeX compensates to some extent with a single alignment column and also gives you the option of typesetting the code in a monospaced font. But suppose instead of stopping after lexing the Haskell we also parse it; this gives us a parse tree which we can pretty-print with table-like environments to ensure that everything is correctly indented.

3 Parsing Haskell

In this section we will start off by considering the high level structure of the parser as a whole. We then take a look at the evolution of each module separately, working on the assumption the input is a single code block, and conclude with the changes that were necessary to parse a module split into several sections as literate programs generally are.

This section does not portray a true to life description of the development process. In practise there was a reasonable amount of overlap in the development of the various phases, but the description offered here gives the best compromise between clarity of description and reality.

3.1 Parser structure

This project can be broken into two halves—parsing Haskell and pretty-printing the abstract syntax tree in LaTeX. Let us start with the first of these, parsing Haskell. Our parser will follow the structure of the Haskell 98 report as closely as possible so as to make it easy to compare the code to the report when debugging or applying fixes made to the report to the code.

As noted in Section 2.2, the job of parsing is in general easiest done when broken into two parts—first the input is lexed, then it is syntactically analysed. The Haskell 98 report has been written with this in mind, and as such a "lexical syntax" and a "context free syntax" can be found in appendix B of the report. As the intention is to follow the structure used in the report as closely as possible, and given the many computer scientist years of experience in this field, we will take this approach.

However, there are three things we have not yet managed to account for. First is the need to extract the executable code from the surrounding text. Although this project is intended to be used for scripts using the second literate style described in the report [27], it is desirable to also cope with the first style and plain Haskell[5]. As this defines the characters the lexer operates on we will do it before the lexing stage.

Second, as in many languages, Haskell allows us to set the associativity and binding precedence (together refered to as the fixity) of infix operators. Fixity information refers to operators rather than individual characters so common sense tells us to handle it after the lexing stage. As the expression "$a \oplus b \otimes c$" produces different parse trees depending on the fixities of $\oplus$ and $\otimes$ we had better handle it before the syntax analysis stage.

The third is less common. Haskell allows some of the structure of a script to be conveyed either by explicitly entering braces and semicolons or with the positions of braces and semicolons implied by the layout of the script; i.e., the whitespace in a Haskell script can be significant. Both styles can be mixed in the same script and the so-called "layout rule" describes where the implicit braces and semicolons go. This description is in terms of tokens such as **where** so it makes sense to do it after the lexer, but the syntax analyser will need to know about the structure of the script, so we must apply the layout rule between the two.

Thus we want to handle both fixity and layout between the lexer and parser, which leaves the question of which we do first. Consider how the scoping of fixity information works in Haskell; a fixity definition applies to the operator of the same name defined in the same declaration group[6]. This operator definition, and thus also the fixity information associated with it, is then used in the current block and all subblocks until a more local declaration of an operator with the same name replaces it. As the fixity information needs to know where the blocks are it is only logical to apply the layout rule first.

[5] Also sometimes refered to as illiterate Haskell.

[6] Two declarations are in the same declaration group if there is a block (a list of definitions surrounded by { and }) directly containing both declarations (i.e. they are not in a nested group).

Now we have a high level overview for what is conceptually[7] a five pass parser:

$$parser = syntax\_analyse \circ fixity \circ layout \circ lex \circ extract\_code$$

We will implement each stage as a module, applying the principles of abstraction and separation of concerns liberally. Where appropriate stages will be further broken down into sub modules, and some modules may be shared between different stages.

3.2 Unlitting

Ultimately the aim is for the parser to be able to cope with multiple sections of code, either in sections delimited by `\begin{code}` and `\end{code}` or blocks of lines beginning with a `>`, and substitute nicely typeset equivalents with the surrounding text untouched.

However, for a first approximation we will simply unlit the script, i.e., extract all the code as a single chunk and throw away the surrounding text. The report also requires that a line containing only white space be between any surrounding text and a program line beginning with a '`>`' character. For now we will ignore this restriction, returning to it when we write the final implementation of this phase.

This simplified process can be fairly straightforwardly coded as shown in Figure 4.

3.3 Parser Combinators

3.3.1 Existing PC libraries

In order to write the lexer we shall use Parser Combinators (PCs). There are two existing PC libraries for Haskell available, UU PC lib and parsec. However, both of these use predictive parsing in order to avoid backtracking and guarantee $O(n)$ time complexity for a given parser on an input of length n. While this is an admirable quality, it means that the set of grammars they allow has to be more restrictive than we would like.

For example, consider the slightly simplified extract from the Haskell lexical syntax in Figure 5(a). Upon finding a '`0`' in the input a predictive parser built directly from this grammar would immediately commit to the first production which allowed a '`0`', here *decimal*. This means that it will never recognise an octal or hexadecimal constant! We can translate this grammar into an equivalent one suitable for use with a predictive parser; such a grammar is shown in Figure 5(b) and algorithms exist to perform such a translation for any grammar.

However, the resulting grammar would bear relatively little resemblance to the grammar in the report which would make it difficult to check its correctness by hand and to update it with new features or fixes. Another example is shown in Figures 6(a) and 6(b). We must also remember that these are simplified extracts of just a small part of the lexical syntax—the effect would be far worse if we were considering it in its entirety. Furthermore we have not gained anything in terms of time complexity, as we shall now demonstrate.

[7] As Haskell is a lazy language the evaluation order will not actually mean the five stages get run sequentially, but for most purposes we can treat it as if they were.

```
unlit :: String → String
unlit = concat ∘ unlit_lines ∘ lineify

unlit_lines :: [String] → [String]
unlit_lines []             = []
unlit_lines ((‘>’:xs):xss) = (‘ ’:xs):unlit_lines xss
unlit_lines (xs:xss)
    | take 12 xs == “\\begin{code}” = if ok
                                     then block ++ unlit_lines rest
                                     else  fail
    where (block, rest) = get_block xss
          ok = all isSpace $ drop 12 xs
          fail = error “\\begin{code} not followed by whitespace”
unlit_lines (_:xss)        = unlit_lines xss

get_block :: [String] → ([String], [String])
get_block []       = error “\\end{code} missing”
get_block (xs:xss)
    | take 10 xs == “\\end{code}” = if ok
                                   then ([], xss)
                                   else  fail
    where ok = all isSpace $ drop 10 xs
          fail = error “\\end{code} not followed by whitespace”
get_block (xs:xss) = (xs:block, rest)
    where (block, rest) = get_block xss

lineify :: String → [String]
lineify “” = []
lineify xs = this:lineify rest
    where (this, rest) = get_line xs

get_line :: String → (String, String)
get_line “”       = (“”, “”)
get_line (‘\n’:s) = (“\n”, s)
get_line (‘\r’:s) = case s of
                      ‘\n’:s’ → (“\r\n”, s’)
                      _ → (“\r”, s)
get_line (c:s)    = (c:ys, zs)
    where (ys, zs) = get_line s
```

Figure 4: *unlit* first draft code

$$\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{decimal} \\
 & | & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} \\
 & | & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal} \\
\mathit{decimal} & \rightarrow & \mathit{digit}\ \{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\ \{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\ \{\mathit{hexit}\} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{octit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{7} \\
\mathit{hexit} & \rightarrow & \mathit{digit} \mid \texttt{A} \mid \ldots \mid \texttt{F} \mid \texttt{a} \mid \ldots \mid \texttt{f}
\end{array}$$

(a) Grammar in Haskell 98 report

$$\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{nzdigit}\ \mathit{decimal'} \mid \texttt{0}\ \mathit{integer'} \\
\mathit{integer'} & \rightarrow & \mathit{decimal'} \\
 & | & \texttt{o}\ \mathit{octal} \mid \texttt{O}\ \mathit{octal} \\
 & | & \texttt{x}\ \mathit{hexadecimal} \mid \texttt{X}\ \mathit{hexadecimal} \\
\mathit{decimal'} & \rightarrow & \{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\ \{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\ \{\mathit{hexit}\} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{nzdigit} & \rightarrow & \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9} \\
\mathit{octit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{7} \\
\mathit{hexit} & \rightarrow & \mathit{digit} \mid \texttt{A} \mid \ldots \mid \texttt{F} \mid \texttt{a} \mid \ldots \mid \texttt{f}
\end{array}$$

(b) Predictive grammar

Figure 5: (Simplified) lexical syntax for integers

$$\begin{array}{lcl}
\mathit{reservedid} & \rightarrow & \texttt{if} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \mid \texttt{instance}
\end{array}$$

(a) Grammar in Haskell 98 report

$$\begin{array}{lcl}
\mathit{reservedid} & \rightarrow & \texttt{i}\ \mathit{reservedid\_i} \\
\mathit{reservedid\_i} & \rightarrow & \mathrm{f} \mid \mathrm{n}\ \mathit{reservedid\_in} \\
\mathit{reservedid\_in} & \rightarrow & \texttt{fix}\ \mathit{reservedid\_infix} \mid \texttt{stance} \mid \epsilon \\
\mathit{reservedid\_infix} & \rightarrow & \mathrm{l} \mid \mathrm{r} \mid \epsilon
\end{array}$$

(b) Predictive grammar

Figure 6: Subset of lexical syntax for reserved IDs

This table shows the time taken for a machine with a 733MHz Intel Celeron CPU and 512MB of RAM to lex 8, 16 and 32 copies of a Haskell script concatenated together respectively.

Size	**CPU time taken**
355k	0m44.240s
710k	1m29.760s
1420k	3m07.750s

Figure 7: Experimental data: Lexical analysis times

Assume that checking that the next input character matches a given symbol can be done in constant time and that expanding a non-terminal into the union of its productions can also be done in constant time. This means that we can also compare the next input character against an arbitrary, but fixed, number of symbols in constant time. This is sufficient to show that *octit*, *digit* and *hexit* take constant time for input of any length.

Now consider *decimal*; each time we try to match an input character with *digit* it takes us constant time as we have shown. If it succeeds then we repeat with one fewer input characters, while if it fails then we stop. Therefore, given an input of length n, we can do a constant amount of work at most n times, so the amount of work we do is $O(n)$. A similar argument shows that *octal* and *hexadecimal* are $O(n)$.

Given a parser for `0o` *octal* and input of length n we would need constant time to match the first character against '`0`', constant time to match the second character against '`o`' and the remaining $n-2$ characters will take $O(n-2)$ time. Therefore the whole parser is also $O(n)$. Similar arguments hold for the other alternatives of the *integer* non-terminal.

Finally we consider *integer*. Each of the alternatives will take $O(n)$ time so we can do all five in $O(n)$ time, proving our assertion that we can lex integers in linear time even though our parser combinators can backtrack.

In order to show that parsing a given grammar using backtracking is linear it is sufficient to show that none of the right hand sides of the productions for a non-terminal α can, after expanding an arbitrary number of further productions, contain an α. In other words it is sufficient for the graph of the productions to be acyclic. We can see that this is the case for the lexical syntax in the report, so we will be able to lex it in linear time using a backtracking PC library. This is backed up by experimental data from the finished lexer shown in Figure 7. An alternative method for fast lazy lexing is explored in [7].

Parsec does have a *try* primitive which relaxes the restrictions imposed by predictive parsing to an extent. However, it also means we lose the guaranteed linear time and the code is no longer as clean. All things considered the best approach seems to be to write our own parser combinator library specially designed for our task.

3.3.2 Preparing for our own PC library

When writing our parser combinator library we will borrow many of the names from the UU PC library. In some cases this will mean the same function or operator has slightly different semantics in the two libraries, in particular the Choice operator. We hope that this will not cause too much confusion.

Before we start to think about implementing our primitives there is one more key point we must pick up from the report.

> In both the lexical and the context-free syntax, there are some ambiguities that are to be resolved by making grammatical phrases as long as possible, proceeding from left to right (in shift-reduce parsing, resolving shift/reduce conflicts by shifting). In the lexical syntax, this is the "maximal munch" rule. In the conText-free syntax, this means that conditionals, let-expressions, and lambda abstractions extend to the right as far as possible.

This means that we also need to know how many input symbols our parsers consume so that, when we have the choice of two parsers which both succeed, we can pick the one which consumes the most input.

3.3.3 Our own PC library: First draft

The first step in the actual implementation of the first draft of our PC library is to consider what the type of a parser should be. We use a type synonym partly to simplify error messages given by our compiler, but mainly because it means we have less to change as the system evolves. In particular, the derived parsers will not need to be changed at all as we add functionality to the primitives.

The minimum information a parser needs to provide us with is whether it succeeded and, if it did, what it returned and how much of the input it consumed. We use the Maybe monad to indicate success or failure and, for convenience, also return the rest of the input. Thus

type Parser α = String → Maybe (α, Int, String)

is a (very simple) first approximation. However, we will want to use our library for both lexing and syntax analysinging and, for the latter, the input will generally not be a list of characters but a list of tokens. Thus we generalise this slightly to a type synonym taking two type variables, one for the input type and one for the output type:

type Parser α β = [α] → Maybe (β, Int, [α])

This done we start to define our primitives, referring to Section 2.3, the existing implementations [24, 29] and the tutorial [15] for an idea of what we want to be able to do.

We start with the Fail and Succeed primitives which are trivial to define, as shown in Figure 8. We pause only briefly to note that *pSucceed* consumes 0 characters.

We note that Match is not primitive in the UU parsing combinators [29], but is defined in terms of the <..> operator, which succeeds if the next symbol is between its two operands inclusive, as defined by the Ord operator $\leq$. The only places where this looks like it might be useful are where we match the upper and lower case letters, digits, octits and hexits. Even then the prelude supplies us with predicates *isUpper*, *isLower*, *isDigit*, *isOctDigit* and *isHexDigit*. Thus a more logical choice of primitive for us is *pPred* which, given a predicate, returns a parser that succeeds if and only if there is at least one more symbol and the

```
pFail :: Parser α β
pFail = λ inp → Nothing

pSucceed :: β → Parser α β
pSucceed v = λ inp → Just (v, 0, inp)
```

Figure 8: *pFail* and *pSucceed* first draft

```
pPred :: (α → Bool) → Parser α α
pPred f = λ inp → case inp of
                      [] → Nothing
                      (t:ts) → if f t
                                 then Just (t, 1, ts)
                                 else  Nothing
```

Figure 9: *pPred* first draft

predicate holds for that symbol. If successful it consumes that symbol, removing it from the input and returning it. The fairly straightforward code is shown in Figure 9.

We will use the application style of Sequential Composition described in Section 2.3. A flowchart describing the operator is given in Figure 10 and can be simply translated into the code in Figure 11.

Our definition of choice operator is influenced strongly by the Haskell report which, as quoted earlier, says "ambiguities that are to be resolved by making grammatical phrases as long as possible" [27, appendix B, section 1]. In other words, if both parsers succeed we take the one that consumed the most input. If both fail then we must also fail and, as you would expect, if only one succeeds then we successfully return the value it returned. Of course, it is possible for both parsers to succeed *and consume the same amount of input.* However, the report does not allow this to happen, so if it does then we fail. The code is given in Figure 12.

By default the operators <∗> and <|> will be considered left associative and have the highest fixity, i.e., they will bind as tightly as possible. As <∗>

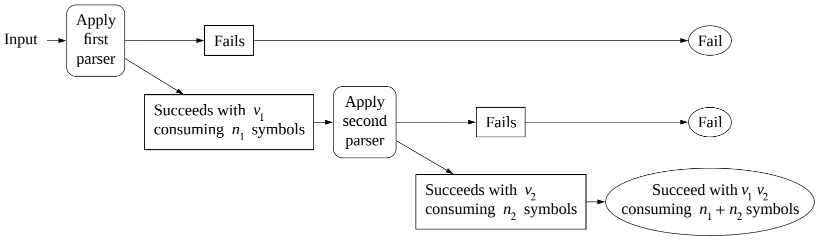

Input
Apply first parser
Fails
Fail
Succeeds with v_1 consuming n_1 symbols
Apply second parser
Fails
Fail
Succeeds with v_2 consuming n_2 symbols
Succeed with v_1 v_2 consuming $n_1 + n_2$ symbols

Figure 10: Flowchart describing sequential composition of parsers

```
(<*>) :: Parser α (β → γ) → Parser α β → Parser α γ
p1 <*> p2 = λ inp →
                 case p1 inp of
                        Nothing → Nothing
                        Just (v1, n1, inp1) → case p2 inp1 of
                                                   Nothing → Nothing
                                                   Just (v2, n2, inp2) → Just (v1 v2, n1 + n2, inp2)
```

Figure 11: Code for sequential composition of parsers

```
(<|>) :: Parser α β → Parser α β → Parser α β
p1 <|> p2 = λ inp →
                 case (p1 inp, p2 inp) of
                        (Nothing, r2) → r2
                        (r1, Nothing) → r1
                        (r1@(Just (_, n1, _)), r2@(Just (_, n2, _))) → case n1 `compare` n2 of
                                                                            GT → r1
                                                                            LT → r2
                                                                            EQ → error same
      where same = "Both operands of <|> consumed same number of symbols"
```

Figure 12: Code for choice of parsers

is roughly analogous to function application it makes sense for it to be left associative for the same reasons that juxtaposition is. The Choice operator is commutative so it makes no difference whether <|> is left or right associative; we arbitrarily decide to make it left associative. It makes sense for one of the operators to bind tighter than the other to reduce the number of brackets required. By making <∗> bind tighter we would allow ourselves to easily write a choice of multiple parsers and would also be following the same precedence rules as BNF, so we choose to do this. To allow space for derived operators inbetween we define the binding precedence of <∗> to be 7 and of <|> to be 3.

This completes the first draft of our PC primitives. Before we test it we will find it convenient to augment our library with a pair of derived functions. The first is the *pSym* function, defined in terms of *pPred* as discussed earlier. Second we observe we will often want to apply a fixed data constructor or function to the result of one or more parsers; we define <$> so that we do not need to clutter up our code with many *pSucceed* applications. Ths code for both is shown in Figure 13.

As the Haskell lexical syntax includes constructs like lists of digits we would be foolish to attempt to construct a lexer before implementing some derived library functions to deal with such constructs; however, it is still wise to test what we have done thus far so that we can check that our library is behaving as expected. The test session is shown in Figure 14. As we can see the parser is correctly accepting an 'a' followed by either a 'b' or a 'c' and returning the pair of what it consumes if it is successful.

Note that Figure 14 shows that two scripts have been loaded, `PCbase.lhs` and `PC.lhs`. All the functions which need to know how a Parser α is constructed,

$$pSym :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{Parser}\ \alpha\ \alpha$$
$$pSym = pPred \circ (==)$$

$$(<\$>) :: (\beta \rightarrow \gamma) \rightarrow \mathsf{Parser}\ \alpha\ \beta \rightarrow \mathsf{Parser}\ \alpha\ \gamma$$
$$f <\$> p = pSucceed\ f <*> p$$

Figure 13: Code for *pSym* and <$>

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
PCbase.lhs
PC.lhs
Type :? for help
PC> ((,) <$> pSym 'a' <*> (pSym 'b' <|> pSym 'c')) "abcde"
Just (('a','b'),2,"cde")
PC> ((,) <$> pSym 'a' <*> (pSym 'b' <|> pSym 'c')) "ac"
Just (('a','c'),2,"")
PC> ((,) <$> pSym 'a' <*> (pSym 'b' <|> pSym 'c')) ""
Nothing
PC> ((,) <$> pSym 'a' <*> (pSym 'b' <|> pSym 'c')) "q"
Nothing
PC> ((,) <$> pSym 'a' <*> (pSym 'b' <|> pSym 'c')) "a"
Nothing
```

Figure 14: Testing first draft basic PC library

i.e., the primitives, are in `PCbase.lhs`. The derived functions are all in `PC.lhs`, which imports and re-exports all of `PCbase.lhs`. This abstraction means that the set of functions we need to check when altering how the parsers work is exactly the set of functions in a single file and, while we currently trust the derived functions not to peek at the internals of the Parser type, when we later hide the implementation details with a datatype whose constructors are not exported, the derived functions, and parsers built with the library, will be *forced* to obey the published interface.

3.3.4 Our own PC library: Augmenting the first draft

If we are to use the PCs then we will need to know whether our parser was successful, what the return value was and whether or not it consumed the entire input. For the last we instead provide a function which returns the remaining input to allow more flexibility. All three require knowledge of the internals of the Parser type so must be placed in `PCbase.lhs`. The code for each is straightforward; see Figure 15.

As mentioned earlier there are many more constructs, such as lists, that will be useful to us as we build our lexer and syntax analyser. Before we go any further let us augment our library by adding more derived parsers and, when necessary, more primitives.

Two more derived parsers, <$ and <$*$, from the UU PC lib are equivalent to <$> and <$*$> respectively except that they ignore the result of the parser in their second argument. They can trivially be defined in terms of existing functions. Both use *const* with the existing function to drop the value of the

```
parse_succeeded :: Maybe (β, Int, [α]) → Bool
parse_succeeded (Just _)  = True
parse_succeeded Nothing = False

ret_val :: Maybe (β, Int, [α]) → β
ret_val (Just (ret, _, _)) = ret
ret_val Nothing            = error "Tried to take ret_val of a failed parse"

rest_of_input :: Maybe (β, Int, [α]) → [α]
rest_of_input (Just (_, _, rest)) = rest
rest_of_input Nothing             = error
                                       "Tried to get rest_of_input of a failed parse"
```

Figure 15: Functions for information about a parse

```
(<$) :: β → Parser α γ → Parser α β
p <$ q = const p <$> q

(<*) :: Parser α β → Parser α γ → Parser α β
p <* q = const <$> p <*> q
```

Figure 16: Code for <$ and <*

second operand; the code is shown in Figure 16 and an example showing how they can be used to drop parentheses matched is in Figure 17.

In addition to *pSym* we define *pSyms*, which matches multiple symbols; it can also be easily defined as a derived function. It takes a list of symbols and produces a parser which accepts the complete list of symbols in order. To achieve this we fold a function which uses *pSym* to parse each individual symbol and constructs a list from the results. A requirement that the type α be an instance of the Eq class is inherited from *pSym*. The code is shown in Figure 18.

We will also want to parse lists of construct. To produce a list we repeatedly apply a parser to the input until it fails, returning all the successful results in a list. Note that we must ensure that the parser cannot succeed without consuming any input or it would succeed forever![8] To prevent this we first try applying the parser to the empty input list and, using *parse_succeeded* to check, give an error if it succeeds.

As well as a parser, our list function will take an Int as an argument. This specifies a minimum number of successful parses we require, i.e., the minimum

[8] In actual fact the way we implement it means we would get an error from <|> as both of the operand parsers will successfully consume the same number, i.e., 0, symbols.

```
p_paren_string :: Parser Char String
p_paren_string = id <$ pSym '(' <*> pList 0 (pPred (')' ≠)) <* pSym ')'
```

Figure 17: Example demonstrating the use of <$ and <*

```
pSyms :: (Eq α) ⇒ [α] → Parser α [α]
pSyms = foldr (λ s p → (:) <$> pSym s <*> p) (pSucceed [])
```

Figure 18: Code for *pSyms*

```
accepts_empty :: Parser α β → Bool
accepts_empty p = parse_succeeded (p [])

opt :: Parser α β → β → Parser α β
p ‘opt‘ v = p <|> if accepts_empty p
                  then pFail
                  else  pSucceed v

pList :: Int → Parser α β → Parser α [β]
pList _ p
    | accepts_empty p = error “Tried to make a list of empties”
pList 0 p       = pList 1 p ‘opt‘ []
pList (i +1) p = (:) <$> p <*> pList i p
```

Figure 19: Code for *pList* and helper functions *accepts_empty* and *opt*

length of the resulting list. While this is non-zero we apply the parser and cons the result to a recursive call requiring one fewer elements. The list with at least 0 elements is the list with at least 1 element or the empty list. This algorithm can be straightforwardly converted to code, producing the derived function shown, along with the (also derived) helper functions *accepts_empty* and *opt*, in Figure 19. We use *opt* as an infix operator; we have given it a binding precedence of 4, so the only operator it binds tighter than is <|>, and made it non-associative as neither left nor right associativity make sense. The guard against the parser successfully consuming 0 characters in *opt* is not necessary for *pList* but we will see that it is useful elsewhere shortly.

These simple lists are just the tip of the iceberg, however. There are several constructs in the Haskell context free grammar which are a list of some construct with another “separator” construct interspersed; a final separator is sometimes allowed at the end of the list. As well as a parser for the “separators” and a parser for the “things” we will also pass a minimum number of “things” we require, as in *pList*, and a boolean value which is True iff a trailing separator is allowed.

If $i + 1$ “things” are required and i is at least 0 then we require a “thing” followed by a normal list of at least i separators followed by “things”. Finally we can have an optional trailing separator if we are allowed. It is here that the extra guard in *opt* mentioned earlier becomes necessary in case the separator parser can successfully consume no input.

If 0 “things” are required then we either act as if 1 “thing” was required, match a single separator if a trailing separator is allowed or simply return the empty list. We only allow the last case if the parser for “things” cannot successfully consume 0 characters; otherwise it is possible that the first and last cases

```
pSList :: Bool → Int → Parser α β → Parser α γ → Parser α ([Either β γ])
pSList _ _ p_s p_t
    | accepts_empty p_s && accepts_empty p_t = error
                                           "p_s and p_t can't both accept empty in pSList"
pSList end 0 p_s p_t      = pSList end 1 p_s p_t <|>
                            ⎛if end                      ⎞
                            ⎜   then wrap ∘ Left <$> p_s ⎟ <|>
                            ⎝   else  pFail              ⎠
                            if accepts_empty p_t
                               then pFail
                               else  pSucceed [ ]
pSList end (i +1) p_s p_t = (λ h mid t → h ++ concat mid ++ t) <$>
                            (wrap ∘ Right <$> p_t) <∗>
                            (pList i ((λ s t → [Left s, Right t]) <$> p_s <∗> p_t))
                            <∗> if end
                                     then wrap ∘ Left <$> p_s ‘opt‘ [ ]
                                     else  pSucceed [ ]

get_rights :: [Either α β] → [β]
get_rights [ ]            = [ ]
get_rights (Left _:es)    = get_rights es
get_rights (Right r:es)   = r:get_rights es
```

Figure 20: Code for *pSList* and *get_rights*

would both successfully consume 0 characters.

The final thing to note is that, before anything else is considered, we first check that the separator parser and the "thing" parser cannot both successfully consume 0 symbols. While we could let *pList* check this for us we can give a more useful error message by doing so ourselves. The complete code is shown in Figure 20. When the "separator" is a semicolon we want the value returned by the parser as it will tell us if the semicolon was inserted by the layout rule or explicitly by the programmar; however, if the "separator" is just a comma, which is always explicitly inserted, then we do not care about the value. We define a functions *get_rights*, shown in the same figure, which we use when we do not care about the values of the separators.

We will need to be able to do things such as excluding keywords from the parser for function names. To this end we define a new primitive operator <!> such that p <!> q succeeds with the value returned by p if q either fails or succeeds but consumes less input; otherwise it fails. The code is given in Figure 21. As we will generally want to exclude a set of parses from an entire parser we give <!> a precedence of 2, the lowest of all our operators, and decide to make it left associative as successive restrictions seem more likely than a restricted restriction.

Two more small functions remain before we have finally completed our first draft, both to do with optional constructs. The first takes a predicate and consumes the first input character if the predicate holds for it, returning True; otherwise it just returns False. The second applies a parser and, if it succeeds,

```
(<!>) :: Parser α β → Parser α γ → Parser α β
p <!> q = λ inp → case (p inp, q inp) of
                       (p_res, Nothing) → p_res
                       (Nothing, _) → Nothing
                       (Just sp@(_, np, _), Just (_, nq, _))
                           | nq ≥ np → Nothing
                           | otherwise → Just sp
```

Figure 21: Code for <!>

```
pPredHolds :: (α → Bool) → Parser α Bool
pPredHolds f = True <$ pPred f ‘opt‘ False

pMaybe :: Parser α β → Parser α (Maybe β)
pMaybe p = if accepts_empty p
           then error "pMaybe passed parser that accepts empty"
           else  Just <$> p ‘opt‘ Nothing
```

Figure 22: Code for *pPredHolds* and *pMaybe*

returns the result wrapped up with the Just data constructor; otherwise it just returns Nothing. It first checks that the parser passed cannot succeed without consuming any input in order to try to protect us from ourselves; if this is not the case then pMaybe can be replaced by Just <$>. The code for both is given in Figure 22.

3.3.5 Error handling: knowing where we are

Currently, if we come across an error we just fail. There are two good reasons for providing more information on where the error occurred; the first is that people using a parser built upon our library will want to know where the errors in their input are so that they can fix them. Secondly, and perhaps more important from our point of view, we will find error reporting useful as a means of finding errors in the parsers we build when we find they fail with valid input! We will provide the user with the line and character where we find the first error which should provide sufficient information for them to be able to pinpoint and correct the problem. Used repeatedly all bugs, whether in the parser combinator library, the parser built on it or the input being parsed, can be fixed.

A position in a script is either a line and character number or the special position "end of file". A sufficient definition of Position would be:

```
type Position = Maybe (Int, Int)
```

with Nothing being used to represent the end of the file but, in the interests of data abstraction and clarity of data constructors, we will create our own datatype and type synonyms:

```
type Line = Int
type Character = Int
```

```
 instance Ord Position
     where (Position l1 c1) ≤ (Position l2 c2) = (l1, c1) ≤ (l2, c2)
           _ ≤ End_Of_File                     = True
           End_Of_File ≤ _                     = False

furthest_pos :: Position → Position → Position
furthest_pos = max
```

Figure 23: Code for *furthest_pos* and definition of Ord Position

```
end_of_file :: Position
end_of_file = End_Of_File
```

Figure 24: Code for *end_of_file*

```
data Position = Position !Line !Character
              | End_Of_File
    deriving Eq
```

The '`!`'s specify that the Position constructor is strict in its arguments; this does not affect the functionality but, as profiling shows, improves the performance.

If we have two positions of possible errors, perhaps from both operands of a Choice failing, then the chances are the error is at the position furthest into the file—for example, if we give a parser that matches either "`hello`" or "`world`" the input "`helli`" then it could return an error at either character 5 or character 1, but the error is clearly at character 5. The solution is to define an order on Positions, which we do by making Position an instance of Haskell's Ord class; the reason we made Position derive Eq earlier as it is a prerequisite for the Ord class. We then export a function *furthest_pos* that takes two Positions and returns the further of the two. The code is shown in Figure 23.

Each input character will be paired with its position. The easiest way to both correctly give each character its position and to unlit the input is to do both at the same time; to avoid too much repetition we will not go into the gory details, but Section 3.8 gives an overview of the final implementation. We will also need to refer to the end of the file position at times, for example when reporting an error after running out of input, and thus positions. With our data structure *end_of_file* is very simple, as shown in Figure 24.

3.3.6 Error handling and the PC primitives

A naïve implementation would just have the primitives report failure at the point at which they fail, i.e., the type Parser would be as given in Figure 25 along with the helper function *num*, which extracts the number of characters consumed by a successful parse. The primitives would be altered in the obvious way; Figure 26 gives some examples along with the new helper function, *get_pos*, which gives the position of the start of the input.

Now consider the example in Figure 27, where *posify* takes a list of characters and pairs each one with its position, starting from character 1 on line 1, and we have derived Show on all the data types so we can see the results. The parser

```
type Parser α β = [(α, Position)] → Parsed α β
data Parsed α β = Success β Int [(α, Position)]
                | Failure Position

num :: Parsed α β → Int
num (Success _ n _) = n
num (Failure _)     = error "num: Can't give num of a failure"
```

Figure 25: Naïve error handling Parser type and helper function *num*

used accepts the string "abcd" repeated any number of times and surrounded by braces. To see why this example is relevant we could imagine 'a' to be loosely representative of a function left hand side, 'b' of the equals sign, 'c' of the right hand side and 'd' of a **where** clause; the parser is then loosely representative of a syntax analyser for a block of function definitions.

The first parse succeeds as expected, consuming all the input. However, in the second parse we have left the third 'c' out of the input. Our parser tells us there was an error on the first (and, in this case, only) line at the tenth character, but we want to be told the error is at the 12th character, i.e., the 'd' where we were expecting a 'c'.

To see the problem consider what happens as the string gets parsed. First the '{' is matched and then a list of "abcd"s is required. The first two "abcd"s are successfully matched but the remaining input, "abd}", does not match, so the *pList* function instead returns the empty list. The final parser is then applied to the remaining input and fails to match a '}' so the position of the 'a' is returned as the error point. The key point to note though is that the *pList* function successfully consumed two characters in one of its alternatives before that alternative failed and the other alternative succeeded having consumed no input. Thus what we must do is to remember the furthest point we reach and fail at *even if we succeed*. We are now ready to implement the second draft of our PC library.

3.3.7 Our own PC library: Second draft

We now augment our definition of Parsed from the previous section with a Position in the Success constructor as discussed. This definition, which we will use for the second draft, is shown in Figure 28 along with some helper functions we will use to rewrite the primitives. Note that the existing derived parsers will continue to work once we have updated the primitive parsers—this is a good example of a case when first class modules would be useful—we could pass one of the modules of primitive parsers as a parameter to the derived module.

The updated primitive parsers are given in Figure 29. The first three are straightforward, but the Sequential Composition, Choice and Exclusion operators need slightly more modification to choose the correct position in all cases. Also note that we have used '*seq*' and $! to force evaluation of the furthest position reached; without this the tree of unevaluated thunks consumes a large amount of heap, as discovered by the profiling features of GHC.

Finally note that all the current parsers lose the position information when

```
get_pos :: [(α, Position)] → Position
get_pos []          = end_of_file
get_pos ((_, p):_) = p

pFail :: Parser α β
pFail = λ inp → Failure (get_pos inp)

pSucceed :: β → Parser α β
pSucceed v = λ inp → Success v 0 inp

pPred :: (α → Bool) → Parser α α
pPred f = λ inp → case inp of
                       [] → Failure end_of_file
                       ((t, p):tps) → if f t
                                          then Success t 1 tps
                                          else  Failure p

(<|>) :: Parser α β → Parser α β → Parser α β
p1 <|> p2 = λ inp →
             case (p1 inp, p2 inp) of
                  (Failure pos1, Failure pos2) → Failure (furthest_pos pos1 pos2)
                  (Failure _, r2) → r2
                  (r1, Failure _) → r1
                  (r1, r2) → case num r1 `compare` num r2 of
                                   GT → r1
                                   LT → r2
                                   EQ → error same
    where same = "Both operands of <|> consumed same number of symbols"

(<*>) :: Parser α (β → γ) → Parser α β → Parser α γ
p1 <*> p2 = λ inp →
             case p1 inp of
                  Failure pos1 → Failure pos1
                  Success v1 n1 inp1 → case p2 inp1 of
                                            Failure pos2 → Failure pos2
                                            Success v2 n2 inp2 → Success v n inp2
                                                where v = v1 v2
                                                      n = n1 + n2
```

Figure 26: Naïve error handling parser combinator primitives

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
Position.lhs
PCbase.lhs
PC.lhs
PC> id <$ pSym ’{’ <*> pList 0 (pSyms "abcd") <* pSym ’}’ $ posify "{abcdabcdabcd}"
Success ["abcd","abcd","abcd"] 14 []
PC> id <$ pSym ’{’ <*> pList 0 (pSyms "abcd") <* pSym ’}’ $ posify "{abcdabcdabd}"
Failure (Position 1 10)
```

Figure 27: Naïve error handling demonstration

```
data Parsed α β = Success (β, Int, [(α, Position)], Position)
                | Failure Position
type Parser α β = [(α, Position)] → Parsed α β

num :: Parsed α β → Int
num (Success (_, n, _, _)) = n
num (Failure _)            = error "num: Can’t give num of a failure"

rinp :: Parsed α β → [(α, Position)]
rinp (Success (_, _, r, _)) = r
rinp (Failure _)            = error
                                "rinp: Can’t give remaining input of a failure"

pos :: Parsed α β → Position
pos (Success (_, _, _, p)) = p
pos (Failure p)            = p
```

Figure 28: Second draft Parser type and helper functions

```
pFail :: Parser α β
pFail = λ inp → Failure (get_pos inp)

pSucceed :: β → Parser α β
pSucceed v = λ inp → Success (v, 0, inp, get_pos inp)

pPred :: (α → Bool) → Parser α α
pPred f = λ inp → case inp of
                       [] → Failure end_of_file
                       ((t, p):inp') → if f t
                                         then Success (t, 1, inp', get_pos inp')
                                         else  Failure p

(<|>) :: (Show α) ⇒ Parser α β → Parser α β → Parser α β
p <|> q = λ inp →
          case (p inp, q inp) of
              (Success s1, Success s2) → case num s1 ‘compare‘ num s2 of
                                              GT → succeed s1 (pos s2)
                                              LT → succeed s2 (pos s1)
                                              EQ → Failure (get_pos inp)
              (Success s1, Failure pos2) → succeed s1 pos2
              (Failure pos1, Success s2) → succeed s2 pos1
              (Failure pos1, Failure pos2) → Failure $! fp
                  where fp = furthest_pos pos1 pos2
    where succeed (v, n, inp, pos) pos' = fp ‘seq‘ Success (v, n, inp, fp)
              where fp = furthest_pos pos pos'

(<*>) :: Parser α (β → γ) → Parser α β → Parser α γ
p1 <*> p2 = λ inp →
             case p1 inp of
                 Failure pos1 → Failure pos1
                 Success s1 → case p2 (rinp s1) of
                                  Failure pos2 → Failure $! fp
                                      where fp = furthest_pos (pos s1) pos2
                                  Success s2 → Success (seq_comp s1 s2)
    where seq_comp (v1, n1, _, pos1) (v2, n2, r, pos2) = fp ‘seq‘
                                                            (v1 v2, n1 + n2, r, fp)
              where fp = furthest_pos pos1 pos2

(<!>) :: Parser α β → Parser α γ → Parser α β
p <!> q = λ inp → case (p inp, q inp) of
                       (p_res, Failure _) → p_res
                       (f@(Failure _), _) → f
                       (Success sp, Success sq)
                           | num sq ≥ num sp → Failure (get_pos inp)
                           | otherwise → Success sp
```

Figure 29: Second draft of the primitive parsers

```
keep_pos :: Parser α β → Parser α (β, Position)
keep_pos p = λ inp →
                case p inp of
                     Failure pos → Failure pos
                     Success (v, n, inp', pos) → Success ((v, get_pos inp), n, inp', pos)

(<&>) :: (β → γ) → Parser α β → Parser α (γ, Position)
f <&> p = keep_pos (f <$> p)

(<&) :: β → Parser α γ → Parser α (β, Position)
f <& p = const f <&> p
```

Figure 30: Code for *keep_pos*, <&> and <&

they are applied. This means that once we have lexed the input there is no position information for the syntax analyser and other stages which is clearly unacceptable. We therefore define the primitive function *keep_pos*, as shown in Figure 30, which takes a parser and returns a parser identical[9] except the return value is paired with the position of the start of the input. We also define <&> and <&, also shown in Figure 30, which are equivalent to <$> and <$ respectively except they apply *keep_pos* to the resulting parser; note, however, that *f* <&> *p1* <*> *p2* does not make sense as you can not apply something of type (α, Position) to a value.

3.3.8 Optimisations

While what we have done so far is sufficient, there are a couple of obvious places where we can optimise. First consider the case of a Choice where we know that if one of the choices succeeds the other never will; for example, when we are looking for keywords during lexical analysis if the parser for the keyword **case** succeeds then there is no point in trying the parser for the keyword **do**—not only can it not succeed but its furthest failure point is guaranteed to be closer. We must be careful though—after finding the keyword **in** we must still look for keywords like **instance** and **infix**. Looking for **in** only if we fail to find **instance** is fine, but note that the furthest position reached of the first parser is relevant; for example, the input may have been "`instanx`".

There are many places where this is useful, both in lexical and syntactical analysis; in particular it is often obvious where it could be used within a production. We therefore create a new primitive <|, which is equivalent to <|> except it does not try the right parser if the left succeeds, for performance reasons. The symmetric equivalent |> is defined in terms of <|. Code for both is given in Figure 31.

Another cause of inefficiency is to be found in the *pList* function. Although we know that a list of at least 0 things will always succeed we still wrap the results with Succeed constructors and then unwrap them with pattern matching later. To get around this we break up the definition of Parser and Parsed and

[9] Technically this is not true as *keep_pos* can alter the strictness properties of the parser; for example, *pSucceed v* is not strict in its next argument, but *keep_pos* (*pSucceed v*) is.

```
(<|) :: Parser α β → Parser α β → Parser α β
p1 <| p2 = λ inp →
             case (p1 inp, p2 inp) of
                  (Success r1, _) → Success r1
                  (Failure pos1, Success (v2, n2, inp2, pos2)) → fp ‘seq‘ Success (v2, n2, inp2, fp)
                      where fp = furthest_pos pos1 pos2
                  (Failure pos1, Failure pos2) → Failure $! fp
                      where fp = furthest_pos pos1 pos2

(|>) :: Parser α β → Parser α β → Parser α β
p1 |> p2 = p2 <| p1
```

Figure 31: Code for short circuit Choice operators <| and |>

introduce two new primitives *pList0* and *pListS*. Both match a list of at least 0 things matched by the parser given as their first argument; the difference is that *pListS* returns an SParser which in turn will produce an SParsed, while *pList0* takes this value and wraps it up with Success. Only Parser and *pList0* are exported from the primitives module and we then insert a new definition for *pList* to make use of it. This is all shown in Figure 32.

With the previous optimisation it was clear that we would gain benefit as it means that a lot of the time we will be able to not apply parsers. In this case, though, there is the possibility that a compiler might optimise out the redundant applications meaning all we are doing is obfuscating the code. As Figure 33 shows, though, this optimisation is worthwhile, giving better than a 40% speedup of the completed pretty-printer on a 43k Haskell script.

3.4 The lexer

Now that we have completed our PC library, constructing the lexer is largely a mechanical task. To save us having to remember the types of the smaller lexers we combine to make larger lexers we shall adopt the naming convention in Figure 34.

Our implementation will be faithful to the report in all but one aspect—we will not attempt to implement the productions for Unicode characters shown in Figure 35, instead allowing only the ASCII subset explicitly listed. We are in good company here—of the Haskell implementations only the poorly supported `hbc` supports Unicode, although support is beginning to appear in the more mainstream implementations.

The simpler terminal productions can be written using *pPred* and a predicate from the Char library; see Figure 36 for an example. Figure 37 shows how *pSym* (and, in other cases, *pSyms*) are used to match other terminals and how our <|> operator, or one of its optimised alternatives, is analogous to the | symbol used to represent choice in the report. This analogy also holds for non-terminals, as shown in Figure 38. Within each production the report uses juxtaposition to represent sequential composition which, together with a suitable function, is analogous to our <∗> operator as shown in Figure 39. This figure also shows how we use *pList* with first argument 0 for the $\{X\}$ syntax representing any

```
type SParsed α β = (β, Int, [(α, Position)], Position)
data Parsed α β = Success (SParsed α β)
                | Failure Position
type SParser α β = [(α, Position)] → SParsed α β
type Parser α β = [(α, Position)] → Parsed α β

pList :: Int → Parser α β → Parser α [β]
pList 0 p       = pList0 p
pList (i +1) p = (:) <$> p <*> pList i p

pList0 :: Parser α β → Parser α [β]
pList0 p = λ inp → Success (pListS p inp)

pListS :: Parser α β → SParser α [β]
pListS p = λ inp →
          case p inp of
              Success (v1, n1, inp1, pos1) → fp `seq` (v1:v2, n1 + n2, inp2, fp)
                  where (v2, n2, inp2, pos2) = pListS p inp1
                        fp = furthest_pos pos1 pos2
              Failure pos → ([], 0, inp, pos)
```

Figure 32: Optimised *pList* and the new types needed

Using normal or optimised *pList*	**CPU time taken**
Normal	10m39.840s
Optimised	6m16.180s

Figure 33: Experimental data: Times for complete runs with and without optimised *pList*

```
lexc_* :: Parser Char Char
lexs_* :: Parser Char String
lext_* :: Parser Char [(Token, Position)]
lexts_* :: Parser Char (Token, Position)
```

Figure 34: Naming convention for lexical analyser components

uniWhite	→	any Unicode character defined as whitespace
uniSmall	→	any Unicode lowercase letter
uniLarge	→	any uppercase or titlecase Unicode letter
uniSymbol	→	any Unicode symbol or punctuation
uniDigit	→	any Unicode decimal digit

Figure 35: Unicode lexical analysis productions

```
ascDigit   →   0 | 1 | . . . | 9

lexc_ascDigit :: Parser Char Char
lexc_ascDigit = pPred isDigit
```

Figure 36: Lexical analysis terminal productions implemented with predicates

```
charesc   →   a | b | f | n | r | t | v | \ | ” | ’ | &

lexs_charesc :: Parser Char String
lexs_charesc = wrap <$> lexc_charesc

lexc_charesc :: Parser Char Char
lexc_charesc = pSym ‘a’ <| pSym ‘b’ <| pSym ‘f’ <| pSym ‘n’ <| pSym ‘r’ <| pSym ‘t’
               <| pSym ‘v’ <| pSym ‘\\’ <| pSym ‘”’ <| pSym ‘\” <| pSym ‘&’
```

Figure 37: Other lexical analysis terminal productions

number, including 0, of Xs and how our <!> operator can be used in place of the exclusion syntax $X_{<Y>}$ used in the report. Where the report uses the [X] syntax we have a choice between using <|> operator or ‘*opt*‘; we pick one depending on which fits best in the particular circumstance. Finally we make use of <&> wherever we are producing tokens to keep the position information for the syntax analyser and other later phases; for an example see Figure 40. The type of the complete lexer is thus [(Char, Position)] → [(Token, Position)].

We have some choice as to what tokens we produce. For example, there are three ways we might handle literals: one option is to produce a Literal token which contains the information for either an integer, float, character or string; another option is to have four seperate tokens, one for each type of literal; the third approach is to have tokens for even the contents of strings rather than a single token for the whole string. As we will often want to treat the type of literal differently, for example, when parsing infix declarations we will need to look for integers, we decide against the first option. The last option makes it difficult to handle strings later, so we go for the second option. We use a second data type, StringContents, to describe the contents of a string; for the other literals we make do with a String. Similarly we put the contents of ordinary comments into a single token, with the number of leading dashes as an Int and the rest of the content as a String, but decide the easiest way to represent nested comments is to have tokens for the beginning and end markers and design the lexer such that only WhiteChar, Newline and nested comment tokens appear between them.

There are a few places where we choose to do things slightly differently to the

```
literal   →   integer | float | char | string

lext_literal :: Parser Char (Token, Position)
lext_literal = integer <|> float <|> char <|> string
```

Figure 38: Lexical analysis choice of non-terminals

$$\begin{array}{lll} \textit{varid} & \rightarrow & (\textit{small}\ \{\textit{small} \mid \textit{large} \mid \textit{digit} \mid {}'\})_{<\textit{reservedid}>} \\ \textit{conid} & \rightarrow & \textit{large}\ \{\textit{small} \mid \textit{large} \mid \textit{digit} \mid {}'\} \end{array}$$

```
id_body :: Parser Char Char
id_body = lexc_small <| lexc_large <| lexc_digit <| pSym '\''

lexs_varid :: Parser Char String
lexs_varid = (:) <$> lexc_small <*> pList 0 id_body <!> lext_reservedid

lexs_conid :: Parser Char String
lexs_conid = (:) <$> lexc_large <*> pList 0 id_body
```

Figure 39: Lexical analysis sequential composition and exclusion

$$\textit{opencom} \quad \rightarrow \quad \{\text{-}$$

```
lexs_opencom = NCommentOpen <& pSyms "{-"
```

Figure 40: Remembering the position in the lexical analyser

report. Perhaps the most common is where we merge $X\{X\}$, using just *pList* 1; for an example see Figure 41(a). Also, the semantics of <!> are different to those of the $X_{<Y>}$ syntax used in the report. Our semantics allow us to implement the *ANYseq* production more clearly, as shown in Figure 41(b).

3.5 The layout rule

3.5.1 Indent Marking

The report describes the layout rule as a 2 part process. The first part, whose rules are summarised in Figure 42, can be straightforwardly turned into a piece of code that simply walks through the list adding additional tokens as required. We use an abstract datatype, defined in Figure 43 along with the functions which use it, to remember what we expect next. The externally visible function, *indentmarkify*, just calls *indentmark* with the initial state BOM; this then simply implements the rules on a case by case basis.

3.5.2 The layout rule

Having augmented the list of tokens we apply the function given in pseudo-code in the report, which we have reproduced in Figure 44 for your convenience. The function is called with *L tokens* [] where *tokens* is the augmented token stream generated above. While most of the function could be implemented in a couple of dozen lines of code there is one clause, pointed at by a ☞ in Figure 44, that is significantly harder.

In order to know if a parse error would occur we will use a state machine; for now we will assume the existence of such an automaton which exports the type Automaton_State and 3 functions, whose type signatures are given in Figure 45; *init_states* is the initial state of the automaton at the beginning of a script, *step* takes the current state of the automaton, the next token and returns the new

$$decimal \quad \rightarrow \quad digit\ \{digit\}$$

```
lexs_decimal :: Parser Char String
lexs_decimal = pList 1 lexc_digit
```

(a) Lists

$$ANYseq \quad \rightarrow \quad \{ANY\}_{\langle \{ANY\}\ (opencom|closecom)\{ANY\}\rangle}$$

```
lexts_ANYseq :: Parser Char [(Token, Position)]
lexts_ANYseq = pList 0 (lext_ANY <!> (lexs_opencom <| lexs_closecom))
```

(b) ANYseq

Figure 41: Lexical analysis: Doing better than the report

Given a list of lexemes, as specified by the lexical syntax, augment the list with additional tokens as described by these rules:

- If a **let**, **where**, **do**, or **of** keyword is not followed by the lexeme "{", the token "{n}" is inserted after the keyword, where n is the indentation of the next lexeme if there is one, or 0 if the end of file has been reached.
- If the first lexeme of a module is not "{" or **module**, then it is preceded by "{n}" where n is the indentation of the lexeme.
- Where the start of a lexeme does not follow a complete lexeme on the same line, this lexeme is preceded by "<n>" where n is the indentation of the lexeme, provided that it is not, as a consequence of the first two rules, preceded by "{n}".

The following rules are also in effect while this is being done:

- The "newline" lexeme starts a new line.
- The first column is designated column 1, not 0.
- Tab stops are 8 characters apart.
- A tab character causes the insertion of enough spaces to align the current position with the next tab stop.
- Unicode characters are considered to be of the same, fixed, width as an ASCII character.

Figure 42: The indent-marking rule

```
data IndentState = BOL
                 | BOS
                 | BOM
                 | Normal

indentmarkify :: [(Token, Position)] → [(Token, Position)]
indentmarkify = indentmark BOM

indentmark :: IndentState → [(Token, Position)] → [(Token, Position)]
indentmark BOM []                        = [(IndentToken 0, end_of_file)]
indentmark BOS []                        = [(IndentToken 0, end_of_file)]
indentmark is []                         = []
indentmark Normal (tp@(Newline, _):tps) = tp:indentmark BOL tps
indentmark is (tp@(t, _):tps)
    | is_white_space t = tp:indentmark is tps
indentmark BOM tps@(tp@(t, p):tps')
    | t == Special '{'            = tp:indentmark Normal tps'
    | t == ReservedID "module" = tp:indentmark Normal tps'
    | otherwise                      = this:indentmark Normal tps
    where this = (IndentToken (get_pos_char p), p)
indentmark BOS tps@(tp@(t, p):tps')
    | t == Special '{' = tp:indentmark Normal tps'
    | otherwise          = this:indentmark Normal tps
    where this = (IndentToken (get_pos_char p), p)
indentmark BOL tps@((_, p):_)
    | otherwise = this:indentmark Normal tps
    where this = (IndentLine (get_pos_char p), p)
indentmark _ (tp@(ReservedID r, _):tps)
    | r 'elem' ["let", "where", "do", "of"] = tp:indentmark BOS tps
indentmark Normal (tp:tps)               = tp:indentmark Normal tps
```

Figure 43: The indent-marking code

$$
\begin{array}{llcll}
 & L\ (<n>{:}ts)\ (m{:}ms) & = & \text{`;'}{:}(L\ ts\ (m{:}ms)) & \textbf{if}\ m = n \\
 & & = & \text{`\}'}{:}(L\ (<n>{:}ts)\ ms) & \textbf{if}\ n < m \\
 & L\ (<n>{:}ts)\ ms & = & L\ ts\ ms & \\
 & L\ (\{n\}{:}ts)\ (m{:}ms) & = & \text{`\{'}{:}(L\ ts\ (n{:}m{:}ms)) & \textbf{if}\ n > m \\
 & L\ (\{n\}{:}ts)\ [\,] & = & \text{`\{'}{:}(L\ ts\ [n]) & \textbf{if}\ n > 0 \\
 & L\ (\{n\}{:}ts)\ ms & = & \text{`\{'}{:}\text{`\}'}{:}(L\ (<n>{:}ts)\ ms) & \\
 & L\ (\text{`\}'}{:}ts)\ (0{:}ms) & = & \text{`\}'}{:}(L\ ts\ ms) & \\
 & L\ (\text{`\}'}{:}ts)\ ms & = & parse\text{-}error & \\
 & L\ (\text{`\{'}{:}ts)\ ms & = & \text{`\{'}{:}(L\ ts\ (0{:}ms)) & \\
☞ & L\ (t{:}ts)\ (m{:}ms) & = & \text{`\}'}{:}(L\ (t{:}ts)\ ms) & \textbf{if}\ m \neq 0\ \text{and}\ parse\text{-}error(t) \\
 & L\ (t{:}ts)\ ms & = & \text{t}{:}(L\ ts\ ms) & \\
 & L\ [\,]\ [\,] & = & [\,] & \\
 & L\ [\,]\ (m{:}ms) & = & \text{`\}'}{:}L\ [\,]\ ms & \textbf{if}\ m \neq 0
\end{array}
$$

Figure 44: The layout rule

init_states :: Automaton_State

step :: Automaton_State → Token → Automaton_State

no_such_state :: Automaton_State → Bool

Figure 45: Automaton exported type signatures

state of the automaton and *no_such_state* takes an automaton state and returns true iff that is not a valid state, i.e., the tokens the automaton has been stepped with thus far to get to the state given do not represent a valid prefix of a Haskell script.

The mutually recursive functions *step* and *step'* apply the rules of the layout rule and actually step the automaton, checking a valid state is reached, respectively. The helper function *parse_error* returns True iff the next token would cause a parse error and *im* serves simply to make the code more readable.

3.5.3 The automaton

All that remains is to implement the automaton. There is a large amount of literature on the creation of automata, e.g., [4], but we will settle here for a simple implementation which nevertheless takes linear time with the state transition table which we will provide it. This state transition table is specified by a text file using a BNF-like grammar. Additional syntax is used as we need to be able to convey more information than the report, which is merely informative.

The actual table description is based on the context free grammar in the Haskell report. As we are ignoring the fixity information at this stage there are a couple of places where we can significantly simply things, e.g., the grammar for expressions, shown in Figure 47(a), is quite complex, but this simplification means we can reduce it to that shown in Figure 47(b).

While it would be possible to make only these changes we also choose to eliminate left-recursion and left-factor within each production; this is sufficient to remove all infinite loops and exponential growth from the grammar. Note that this does not make the resulting state machine deterministic as the left-factoring is only local; there are still states from which an opening parenthesis, for example, could move to one of a number of states.

We will think of our automaton as being composed of numerous smaller automata which can link to each other. Each of these automata corresponds to the production of a non-terminal in the context free grammar. When a link between automata is followed a return state within the calling automaton is also given. When we reach the end state of the called automaton we return to the return state in the calling automaton. Our state is therefore a stack of LocalStates, or pairs of integers, identifying a production and position within the production. The top pair on the stack is our current position and the rest of the stack is the history of return addresses. These datastructures are shown in Figure 48.

The first function our automaton must provide is an inital state. Initially we have a single state at the start of the first production and we have no return address. We will guarantee the first production is given number 0 and that the

```
offsideify :: [(Token, Position)] → [(Token, Position)]
offsideify = step init_states []

step :: Automaton_State → [Int] → [(Token, Position)] → [(Token, Position)]
step as ms (tp@(t, _):tps)
    | is_white_space t = tp:step as ms tps
step as (m:ms) tps@((IndentLine n, p):tps')
    | m == n = step' as (m:ms) [im ‘;’ p] tps'
    | n < m   = step' as ms [im ‘}’ p] tps
step as ms ((IndentLine _, _):tps)          = step as ms tps
step as ms@(m:_) ((IndentToken n, p):tps)
    | n > m = step' as (n:ms) [im ‘{’ p] tps
step as [] ((IndentToken n, p):tps)
    | n > 0 = step' as [n] [im ‘{’ p] tps
step as ms ((IndentToken n, p):tps)         = step' as ms [im ‘{’ p, im ‘}’ p]
                                               ((IndentLine n, p):tps)
step as (0:ms) (tp@(Special ‘}’, _):tps)    = step' as ms [tp] tps
step as ms (tp@(Special ‘{’, _):tps)        = step' as (0:ms) [tp] tps
step as (m:ms) tps@((t, p):_)
    | m ≠ 0 && parse_error as t = step' as ms [im ‘}’ p] tps
step as ms (tp:tps)                         = step' as ms [tp] tps
step _ [] []                                = []
step as (m:ms) []
    | m ≠ 0 = step' as ms [im ‘}’ end_of_file] []
step _ (0:_) []                             = error
                                               (“Missing } at ” ++ show_pos end_of_file)

step' :: Automaton_State → [Int] → [(Token, Position)] → [(Token, Position)] → [(Token, Position)]
step' as ms [] tps                = step as ms tps
step' as ms (tp@(t, p):tps) tps'
    | no_such_state as' = error (“No such state at ” ++ show_pos p)
    | otherwise          = tp:step' as' ms tps tps'
    where as' = step_automaton as t

parse_error :: Automaton_State → Token → Bool
parse_error _ (Special ‘}’) = True
parse_error ss t            = no_such_state (step ss t)

im :: Char → Position → (Token, Position)
im c p = (ImplicitSpecial c, p)
```

Figure 46: The layout rule code

$$\begin{array}{lcl}
\mathit{exp}^{i} & \rightarrow & \mathit{exp}^{i+1}\,\left[\mathit{qop}^{(n,i)}\,\mathit{exp}^{i+1}\right] \\
 & | & \mathit{lexp}^{i} \\
 & | & \mathit{rexp}^{i} \\
\mathit{lexp}^{i} & \rightarrow & \left(\mathit{lexp}^{i} \mid \mathit{exp}^{i+1}\right)\,\mathit{qop}^{(l,i)}\,\mathit{exp}^{i+1} \\
\mathit{lexp}^{6} & \rightarrow & -\mathit{lexp}^{7} \\
\mathit{lexp}^{i} & \rightarrow & \mathit{exp}^{i+1}\,\mathit{qop}^{(r,i)}\,\left(\mathit{rexp}^{i} \mid \mathit{exp}^{i+1}\right) \\
\mathit{exp}^{10} & \rightarrow & \ldots
\end{array}$$

(a) Grammar in Haskell 98 report

$$\begin{array}{lcl}
\mathit{exp}^{i} & \rightarrow & [-]\,\mathit{exp}^{10}\,\left[\mathit{qop}\ \mathit{exp}^{i}\right] \\
\mathit{exp}^{10} & \rightarrow & \ldots
\end{array}$$

(b) Simplified grammar

Figure 47: Context free grammar for expressions

```
type Automaton_State = [State]
type State = [LocalState]
type LocalState = (Integer, Integer)
```

Figure 48: Automaton data structures

initial state of each automaton is state 0, so this is just $[[(0,0)]]$ as shown in Figure 49.

Next we must be able to step the automaton, i.e., given a Token t and the current automaton state we must return the new automaton state. To do this we take each of the set of States our automaton could be in and try to step them with t.

The state transition table defines a partial function from LocalStates to StateChanges, a data structure which describes a change of state. Equivalently, $\mathit{stt}\colon \mathsf{LocalState} \rightharpoonup \mathsf{StateChange}$. The data structure used by the automaton to represent state changes is shown in Figure 50 and will be explained shortly.

We can break down the possible state changes into two sets: those which are dependent on the token stream and those which are not. To try to step a State we first apply apply all state changes that are independant of the token stream.

Suppose the top of a State stack s is (h, i), i.e., we are in position i within production h, the remainder of the stack is s' and let $sc = \mathit{stt} \cdot (h, i)$. There are 3 patterns of sc that correspond to token independent state changes:

End This indicates that we have reached the end of this production. The state is thus equivalent to s'.

Jump js This represents a choice to jump to any state $j \in js$. This state is equivalent to the set of states $\{(h, j) : s' \mid j \in js\}$.

Long_Jump j k This represents a jump to another production. We jump to the start state of production j and return to state k in the current production, so this state is equivalent to $(j, 0) : (h, k) : s'$.

```
init_states :: Automaton_State
init_states = [[(0, 0)]]

no_such_state :: Automaton_State → Bool
no_such_state = null

step :: Automaton_State → Token → Automaton_State
step ss t = [ls | (e, ls) ← closure ss, can_step e (un_im t)]

un_im :: Token → Token
un_im (ImplicitSpecial s) = Special s
un_im t                   = t

can_step :: Either Token Func → Token → Bool
can_step (Left s) t               = s == t
can_step (Right (Func (_, f))) t = f t

closure :: [State] → [(Either Token Func, State)]
closure = remove_dupes ∘ sort ∘ do_all_nulls

remove_dupes :: (Eq α) ⇒ [α] → [α]
remove_dupes (a:b:xs)
    | a == b    = remove_dupes (b:xs)
    | otherwise = a:remove_dupes (b:xs)
remove_dupes xs         = xs

do_all_nulls :: [State] → [(Either Token Func, State)]
do_all_nulls []             = []
do_all_nulls ([]:qs)        = do_all_nulls qs
do_all_nulls ((qh:qt):qs) = case lookupFM state_changes qh of
                                Nothing → error "do_all_nulls: Can't happen (Nothing)"
                                Just x → case x of
                                             Jump xs → do_all_nulls (tailed_xs ++ qs)
                                                 where tailed_xs = map ((:qt) ∘ (,) h) xs
                                             Long_Jump i j → do_all_nulls (((i, 0):(h, j):qt):qs)
                                             SLiteral e i → (e, (h, i):qt):do_all_nulls qs
                                             End → do_all_nulls (qt:qs)
    where h = fst qh

state_changes :: FiniteMap LocalState StateChange
state_changes = listToFM state_changes'

state_changes' :: [(LocalState, StateChange)]
```

Figure 49: Automaton code

```
data StateChange = Jump [Integer]
                 | Long_Jump Integer Integer
                 | SLiteral (Either Token (Token → Bool)) Integer
                 | End
    deriving (Eq, Ord)
```

Figure 50: State change data structures

Although we have described equivalences it is only useful to follow the state changes in the forward direction. We apply these rules to each of our possible States until it is not possible to apply them to any State; we also remove any empty stacks as these represent completed traversals of the whole state machine so cannot accept any token t. The function *closure*, shown in Figure 49, implements this, using *do_all_nulls* to do the work and then removing any duplicates in the list of states reached. It returns each of the States as a tuple of the token-dependent test and the State to use if the test succeeds, i.e., if the token would be accepted at the state.

The reason for not using the *nub* function from the List standard library for removing duplicates is that profiling revealed that this was by far the slowest part of the program; this was then replaced as shown with the *sort* function from the standard library but was still responsible for the majority of execution time, so a sorting function using merge sort was written instead. This took the time for the complete pretty-printer to run from 19 minutes, then down to 6 minutes and finally down to 1 minute for a test input.

The next thing we must do is to decide whether the token t is permitted at each of our potential states. There are two tests which we may have to perform:

Left u	This succeeds iff $t = u$.
Right (SLiteral f)	This succeeds iff $f\ t$ holds.

This is implemented by *can_step* in Figure 49.

The stepping function, *step*, returns each possible state just after the point where the token is accepted. This means that even when we have reached the end of the state machine there will be a non-empty stack which will only become empty, and thus removed, when *do_all_nulls* is applied. As neither *init_states* nor *step* apply this to what they return the user of the automaton is returned an empty list if and only if the sequence of tokens stepped so far is not a valid prefix of the language. We can therefore use the standard function *null* for *no_such_state*. The code for this is also given in Figure 49.

The actual table has not been included due to its size. Another program, using the PC library we have developed, parses the textual description of the automatonand produces either the diagrams in Appendix A or the code for the automaton module.

3.6 The fixity annotation

Before we can lexically analyse the token stream we need to add the fixity information so that we can parse the expressions correctly. The details are messy and uninteresting, so we will present only a brief overview here.

For each block we do two passes. In the first pass we walk through, counting our nesting depth to know when we have finished the block, and extract the top-level fixity information—more on that later. Then, in the second pass, we change the fixities of any appropriate tokens, dealing with sub-blocks recursively. There are various things we must take care of such as overriding our fixity list with that of the **where** clause of a declaration where appropriate; although these are all fairly simple, a reasonable amount of code is needed to cover them all. Finally we return the altered block of tokens together with the set of top-level fixity information we collected in the first pass; it is this that is of relevance if this block is a **where** clause.

To extract the fixity information we not only need to look for and extract the information from fixity declarations started with an **infix**, **infixl** or **infixr** token but we also need to notice when new functions are defined and thus override the fixity information from a higher scope. The variety of ways in which functions can be defined adds to the volume of code needed to handle all cases correctly.

The Read instance of Integer makes it easy for us to extract the value of the optional integer given in a fixity declaration. We use *reads* rather than *read* so we can give a useful error message, including position, should a bug in the lexer allow an invalid integer through.

A complete implementation would also look for imported modules and check what these modules export and what restrictions are placed by the import statement. Here we settle for simply starting with the fixity information set by the standard prelude.

This is, in the worst case, quadratic in the length of the input as, for example, adding an extra layer to deeply nested **where** blocks will require traversing all the existing input extra times to handle the outer-most case. In practise the depth of nesting rarely goes above three or four and so this algorithm can be thought of as linear. For better worst-case performance a data structure which allows a token stream to be represented as a tree of blocks could be used, but we believe it to be overkill for real-world use.

3.7 The syntax analyser

The syntax analyser represents a similar challenge to the lexical analyser and the same equivalences of our operators to the report syntax hold. The exclusion syntax $X_{<Y>}$, for which the semantics differ to our <!> operator, appears only once, but we need to handle this case specially. The relevant productions are shown in Figure 51(a); although the report allows a labelled update to be applied to other productions of *aexp* such as tuple this is not actually possible so for simplicity we do not allow these cases. We use *pMaybe* to allow an optional labelled update after these productions; see the code for *qvar* in Figure 51(b) for details.

This is not the only place where we deviate from the report. The production for *body* in the report is shown in Figure 52; the problem is that *impdecls* will consume trailing semicolons as it allows empty import declarations so enforcing the presence of a separating semicolon if there are also *topdecls* is tricky. While we could solve this it would add a significant amount of complexity to the code; by contrast, doing it in another pass, which is necessary[10] anyway to check other side-conditions imposed by the report, is easy.

[10] We do not implement this final pass as it would serve only to restrict the set of valid

$$
\begin{array}{lcl}
\mathit{ANYseq} & \rightarrow & \mathit{qvar} \\
 & | & \ldots \\
 & | & \mathit{qcon}\ \{\mathit{fbind}_1, \ldots, \mathit{fbind}_n\} \\
 & | & \mathit{aexp}_{\langle \mathit{qcon} \rangle}\ \{\mathit{fbind}_1, \ldots, \mathit{fbind}_n\}
\end{array}
$$

(a) Context free grammar

```
p_aexp_qvar :: Parser Token Aexp
p_aexp_qvar = ⎛λ a m_u → case m_u of                   ⎞
              ⎜                  Nothing → a           ⎟ <$>
              ⎝                  Just u → Aexp_update a u⎠
              (Aexp_qvar <$> p_qvar) <*> pMaybe p_aexp_update


p_aexp_update :: Parser Token [[Fbind]]
p_aexp_update = pList 1 p_aexp_update_seg


p_aexp_update_seg :: Parser Token [Fbind]
p_aexp_update_seg = id <$ p_curly_open <*> pList 1 p_fbind <* p_curly_close
```

(b) Code

Figure 51: Exclusion in the context free grammar

$$
\begin{array}{lcl}
\mathit{body} & \rightarrow & \{\mathit{impdecls}; \mathit{topdecls}\} \\
 & | & \{\mathit{impdecls}\} \\
 & | & \{\mathit{topdecls}\}
\end{array}
$$

Figure 52: *body* context free grammar

$$aexp \quad \rightarrow \quad \left(exp^{i+1}\, qop^{(l,i)}\right)$$

Figure 53: *aexp* left section context free grammar

$$btype \quad \rightarrow \quad [btype]\ atype$$

(a) Production

```
p_btype :: Parser Token Btype
p_btype = Btype <$> pList 1 p_atype
```

(b) Code

Figure 54: *btype* production and code

Enforcing the restriction on left sections, whose production is shown in Figure 53, that the operator be of a lower precedence-level than the expression can be done, but the easier ways of doing it are slow, as they require the expression being parsed multiple times, and the fast ways again add complexity. Thus we also leave this check until the next pass, allowing any expression followed by a left-associative operator in parentheses as a left section.

Another issue we must deal with is left recursion. This was not an issue in the lexical syntax, but the context free grammar has productions such as that shown in Figure 54(a) where a *btype* can begin with a *btype* so a direct translation into parser combinators would yield a parser which recursed forever looking for *btype*s. However we can see that a *btype* is just a list of *atype*s, giving us the code shown in Figure 54(b). In general, if we have a production $\alpha \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$, then this is equivalent to the non-left recursive $\alpha \rightarrow (\gamma_1 \mid \ldots \mid \gamma_m)\, \{\beta_1 \mid \ldots \mid \beta_n\}$. In this case $m = 1$, $n = 1$, $\beta_1 = atype$ and $\gamma_1 = \epsilon$.

There are some cases where we implement a slightly different, but equivalent, grammar to the report. For example, if we were to implement the production for *exp* from the report, as shown in in Figure 55(a), then all the expressions would be parsed twice—once for each alternative. We instead implement it as if it was the production in Figure 55(b). This is especially significant as *exp*s can contain *exp*s so the direct translation would have exponential time complexity as opposed to the linear time complexity of this approach. There are more complex examples where left-factoring is needed but the same principles can be applied.

3.8 Coping with multiple code blocks

Up until now we have been extracting all the code from the script and treating it as a single block of code. However, in reality a literate script consists of alternating blocks of explanatory text and Haskell code. We therefore introduce the type

inputs and so is not required for our purposes.

$$\begin{array}{lcl} \mathit{exp} & \rightarrow & \mathit{exp0} :: [\mathit{context} \Rightarrow]\ \mathit{type} \\ & | & \mathit{exp0} \end{array}$$

(a) Report

$$\begin{array}{lcl} \mathit{exp} & \rightarrow & \mathit{exp0}\ [:: [\mathit{context} \Rightarrow]\ \mathit{type}] \end{array}$$

(b) Optimised

Figure 55: Productions for *exp*

```
type Input α = [Either String (String, α, Position)]
```

to hold the input. A Left s represents a text block while a Right (s, x, p) represents a code segment. Depending on the stage of the process we are at, the type of x may be [(Char, Position)], [(Token, Position)] or [Parsed]. The original text of the code block is stored in s; if at some phase an error is encountered then we will replace this block with Left s, giving a warning that we are doing so, and continue the phase at the next code block with the state as it was at the beginning of this block. The position of the end of this block is stored in p; we use this rather than *end_of_file* when we need to give a position and have consumed all of the input in this block.

The first phase, unlitting and adding position information, involves a reasonable amount of code for something that is conceptually quite simple, so we do not give the code here. There are three stages to it: first we read and categorise each line as either starting with a bird track, being part of a `\begin{code}`...`\end{code}` block or being a comment line; we add a position to each character of code. In the second stage we walk through the list checking that all bird track lines are not next to comment lines with any non-whitespace characters. In the final stage we merge the consecutive lines of the same type into blocks.

For the next phase, the lexer, we have to place some restrictions on where we allow code blocks to end. The report allows them to end anywhere a new line is allowed, but it is hard to know how to typeset functions with documentation sections in the middle. We therefore require that top-level functions are entirely contained in a single code block. We also require that nested comments are entirely contained in a single code block; this makes the implementation of the lexer slightly easier at little or no inconvenience to the user. If it proves to be a problem then it would be fairly straightforward to modify the lexer to carry the nesting depth forward from section to section. The lexer then becomes trivial to adapt—we simply apply the existing function to each of the code sections in turn, changing any code section that fails to a comment section.

The indent marking and layout rule phases are also fairly straightforward; while indent marking, when we reach the the end of any code block but the last we stop when our stack of indents reaches a singleton list rather than the empty list so as to not close the outermost braces around all the code. The actual layout rule needs a couple of small tweaks; for example, implicit semicolons at the beginning of code blocks are removed as they cause ugly space at the beginning of all but the first code block.

For the syntactical analysing phase we replace our top-level *syn_an* function with 4 functions: *syn_an_modulehead*, *syn_an_impdecls*, *syn_an_topdecls* and *syn_an_moduletail*. Our initial state is a list of these 4 parsers tupled with a boolean indicating whether they are allowed to multiple, or indeed zero, times; i.e. it is true for only the middle two. While a code block is non-empty we try to apply the first parser in the list; if the list is empty then we give an error. If the parse succeeds and the boolean is false then we remove the parser from the list and recurse with the input remaining after the parse; if the boolean is true then we leave the parser on the list and recurse. If the parse fails and the boolean is true then this is acceptable so we just remove the parser from the list and recurse; if the boolean is false then we do not allow this parser to fail so we abort with an error.

4 Pretty-printing Haskell in LaTeX

Now that we have an abstract syntax tree for Haskell we need to pretty-print it in LaTeX. Rather than embedding the formatting information in the output, with the additional disadvantage of needing to alter the code to change it, we create a style file that defines LaTeX functions and output LaTeX code that calls these functions. A detailed description of all the LaTeX functions is beyond the scope of this report, but we give a brief overview of one of them here.

Figure 56 starts by defining `@layoutinner` and `@layout`, two are internal commands used to set the formatting of paragraphs in many places. A column type, called `Y`, is then defined that is the same as a column of type `X`, defined in the `tabularx` package, but with these layout rules applied. We then have two exported functions; the first, `HtLkeyword`, takes a single argument, presumed to be a Haskell keyword, and typesets it in bold face. The "`HtL`" prefix stands for "Haskell To LaTeX" and is designed to try to avoid name collisions. The final function defined is used to typeset **where** clauses. The first of the three arguments is 0 if braces are not required around the contents of the block or 1 if they are, the second is the contents of the the block and the third is 1 if and only if a semicolon should follow the block. The layout is handled by a `tabularx`, a type of table which will resize columns of type `X`, or `Y` as it is defined in terms of `X`, to make the table the maximum width available; this allows reasonable layouts to be easily achieved.

We then write a Haskell module which works through an abstract syntax tree outputting the appropriate LaTeX, making use of these functions. The module makes use of a few Haskell functions too, for example: a function for pretty-printing the unqualified half of operators checks to see if the operator is one of a list of special operators like `>=` and, if so, outputs special LaTeX for it—in this case `$\ge$`; the escaping of characters necessary in LaTeX is done by `escape`, which is applied to any user-entered strings that are to be output and performs substitutions like `$\backslash$` for `\`; another function substitutes Greek letters for type variables.

```
\def\@layoutinner{%
    \raggedright
    \parindent=0pt%
}
\def\@layout{%
    \everypar{\@layoutinner}\@layoutinner
}

\newcolumntype{Y}{>{\@layout}X}

\def\HtLkeyword#1{%
    \textbf{#1}%
}

\long\def\HtLwhere#1#2#3{%
    \ifcase#1\def\dobracesopen{ }\def\dobracesclose{ }%
          \or\def\dobracesopen{ \{\cr}\def\dobracesclose{\}}\fi
    \begin{tabularx}{\hsize}[t]{@{}r@{ }Y@{}}%
        \hspace*{2em}\IGLhaskellkeyword{where}\dobracesopen
        \@ifmtarg{#2}{}{& #2\cr}%
        \dobracesclose\ifx#31;\fi
    \end{tabularx}%
}
```

Figure 56: Extract from the LaTeX style file

Source size	**CPU time taken**	**Processing speed**
44k	1m20.600s	551 bytes per second
88k	2m43.620s	543 bytes per second
176k	5m31.870s	535 bytes per second
352k	11m14.850s	526 bytes per second
704k	23m24.520s	505 bytes per second

Figure 57: Experimental data: Times and speeds for complete runs on various input sizes with the heap controlled by the run-time system

Source size	**CPU time taken**	**Processing speed**
44k	1m14.170s	598 bytes per second
88k	2m25.540s	610 bytes per second
176k	4m52.170s	608 bytes per second
352k	9m42.200s	610 bytes per second
704k	19m22.840s	611 bytes per second

Figure 58: Experimental data: Times and speeds for complete runs on various input sizes with a fixed 300M heap

5 Discussion

5.1 Evaluation

Now that we have completed the pretty-printer and believe that each phase works in linear time, at least during realistic use, it is time to test our belief. The result of running the pretty-printer on input of various lengths is shown in Figure 57; while the results are very respectable they are not linear. A bit of experimentation shows that the problem is the memory allocation done by the run-time system—if we for the run-time system to use a 300MB heap in all cases by adding `+RTS -M300M -H300M` to the command line then we get the times in Figure 58. Although these appear to show the pretty-printer working in better than linear time this is clearly not actually possible—the cause is the constant startup overhead. The pretty-printer is working at more than $\frac{1}{2}$kB/s on a 733MHz machine, which is fairly modest by today's standards. The files being tested are almost entirely Haskell code—in practise we would expect a literate script to have a high proportion of documentation text which would be dealt with much more quickly, so we would expect real-world usage to be even faster.

The parsing phase of Haskell⤵LaTeX successfully pulls together a number of existing technologies; the result is acceptably fast on modern machines. We have also made it possible to stop between each stage and output either the data structures as shown by the derived Show instances or a simple pretty-printing function. Some sample input together with some of the outputs possible are shown in Figure 59; output is possible at a few additional points, but the pretty-printed version is identical to one of the others.

The LaTeX output phase is more prototypical; although it produces acceptable output for simple functions, more complex functions can lead to poor lay-

```
foo x = y + z
    where y = 2 * x
          z = x + 3
```

(a) Original input

```
{1}foo x = y + z
    <5>where {11}y = 2 * x
          <11>z = x + 3
```

(b) After indent marking

```
{foo x = y + z
    where {y = 2 * x
          ;z = x + 3
}}
```

(c) After layout

```
{foox=y+zwhere{y=2*x;z=x+3}}
```

(d) Parseable tokens

$$\begin{array}{l} \mathit{foo}\ x = y + z \\ \quad \textbf{where}\ y = 2 \times x \\ \qquad\qquad\ \ z = x + 3 \end{array}$$

(e) Code

Figure 59: Output from the pretty-printer at various stages

out. Many of the difficulties faced in the output phase are due to limitations in LaTeX and TeX. They were designed to be used to typeset English text and mathematics, while we require a more precise and restrictive layout. However, there are still acceptable layouts for many of the constructs, but to support more than one layout for a given construct requires a lot of TeX programming; a typesetting language designed with this sort of layout in mind would provide an easier target for a high quality Haskell pretty-printer. One problem with TeX that is very hard to get round is that anything surrounded by automatically sized parentheses or square brackets, such as the case statement in the function below, is never split across multiple lines no matter how wide it is.

$$\mathit{foo}\ x = \left(\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \qquad 0 \rightarrow \text{“hello”}\\ \qquad \_ \rightarrow \text{“goodbye”}\end{array}\right) +\!\!+\ \text{“ world”}$$

Another issue is that LaTeX and TeX allow hand tweaking by the user as they accept that they will not be able to produce the optimum result in all possible cases. This is harder to do in an automated pretty-printer; it would be possible to pass hints and instructions to the pretty-printer by means of Haskell comments, although this would reduce the legibility of the plain source code.

5.2 Other applications

The Haskell parser we have developed is not limited to use in a LaTeX pretty-printer. Many other applications are conceivable, from merely pretty-printing into different languages, e.g., HTML, to program transformation systems like MAG [10], or even a full Haskell interpreter or compiler.

The parser combinator library is also reusable. With a Position module which does not perform the unlitting, parsers for other languages defined by syntax with similar semantics can be written. Indeed, as talked about elsewhere in this report, this author has already written two other parsers using the parser combinator library to aid in the writing of the parser and of this report.

5.3 Problems with Haskell report and implementations

As well as the direct contribution to the Haskell community in the form of a parser and pretty-printer for Haskell, several other contributions have been made along the way. This project was written during a period when the Haskell 98 report was being revised and many typos, errors and inconsistencies were uncovered by this author.

During the implementation of the parser many bugs were found in each of the three major implementations. Some of them were found when they failed to compile or interpret the code for the parser correctly, some were noticed when they did not accept the test-suite written to test the parser and others were found when attempting to confirm the meaning of the report by investigating what the implementations did and how they differed from each other and what had been understood from the report.

Bibliography

[1] *Google.* http://www.google.com/.

[2] *Haskell.* http://www.haskell.org/.

[3] *The LaTeX3 Project.* http://www.latex-project.org/.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[5] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Press, 2nd edition, 1998.

[6] M. Chakravarty. *Haskell Style for LaTeX 2ε.* Available from URL: http://www.cse.unsw.edu.au/~chak/haskell/haskell-style.html.

[7] M. M. T. Chakravarty. Lazy lexing is fast. *Lecture Notes in Computer Science*, 1722:68–84, 1999. Available from URL: http://www.cse.unsw.edu.au/~chak/papers/lexer.ps.gz.

[8] A. Cooke. *haskell.sty.* Available from URL: http://www.andrewcooke.free-online.co.uk/jara/pancito/haskell.sty.

[9] *CTAN.* http://www.ctan.org/.

[10] O. de Moor and G. Sittampalam. *MAG.* Available from URL: http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/mag.htm.

[11] J. K. et al. *Lout.* Available from URL: http://snark.ptc.spbu.ru/~uwe/lout/.

[12] M. Goosens, F. Mittelbach, and A. Samarin. *The LaTeX companion.* Addison-Wesley, 1993.

[13] *Haskell CVS Repository.* http://cvs.haskell.org/cgi-bin/cvsweb.cgi/.

[14] R. Hinze. *lhs2TeX.* Available from URL: http://www.informatik.uni-bonn.de/~ralf/software.html#lhs2TeX.

[15] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996. Available from URL: http://www.cs.nott.ac.uk/~gmh/monparsing.ps.

[16] *Don Knuth's Home Page.* http://sunburn.stanford.edu/~knuth/.

[17] D. E. Knuth. *Appendix to the Errors of TeX paper (updated).* CTAN:/systems/knuth/errata/errorlog.tex [9].

[18] D. E. Knuth. *TeX82.* CTAN:/systems/knuth/tex/tex.web [9].

[19] D. E. Knuth. WEB. Available from URL: CTAN:/systems/knuth/web.tar.gz [9].

[20] D. E. Knuth. Literate programming. *The Computer Journal*, 27(02):97–111, May 1984. Available from URL: `http://www.literateprogramming.com/knuthweb.pdf`.

[21] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1986.

[22] D. E. Knuth. *Literate Programming*. Center For The Study Of Language And Information (CSLI), 1992.

[23] L. Lamport. *LATEX: User's guide & reference manual*. Addison-Wesley, 1985.

[24] D. Leijen. *Parsec*. Available from URL: `http://www.cs.ruu.nl/~daan/parsec.html`.

[25] *Literate Programming*. `http://www.literateprogramming.com/`.

[26] T. Oetiker. *The Not So Short Introduction to LATEX 2ε*. `CTAN:/info/lshort/` [9].

[27] S. L. Peyton Jones and J. Hughes, editors. *The Haskell 98 Report*, 1999. Available from URL: `http://www.haskell.org/definition/`.

[28] N. Porter, editor. *Webster's Revised Unabridged Dictionary*. G & C. Merriam Co., 1913. Available from URL: `http://www.dictionary.com/`.

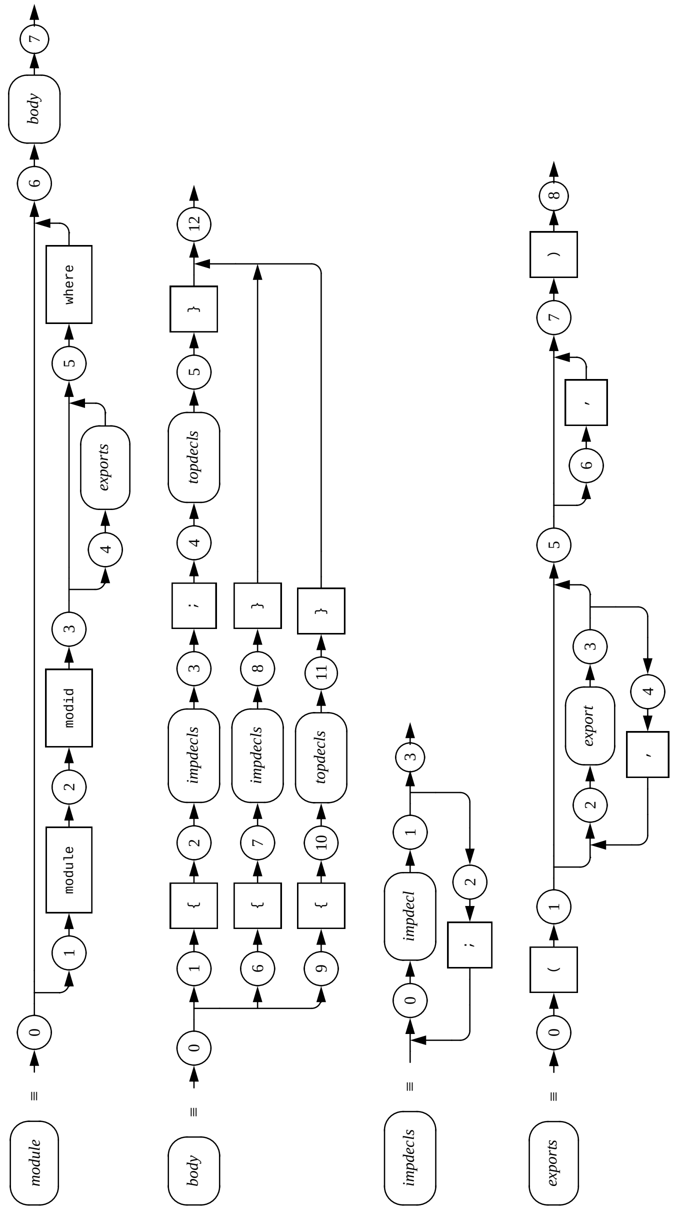
[29] S. D. Swierstra. *Fast, Error Repairing Parsing combinators*. Available from URL: `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/`.

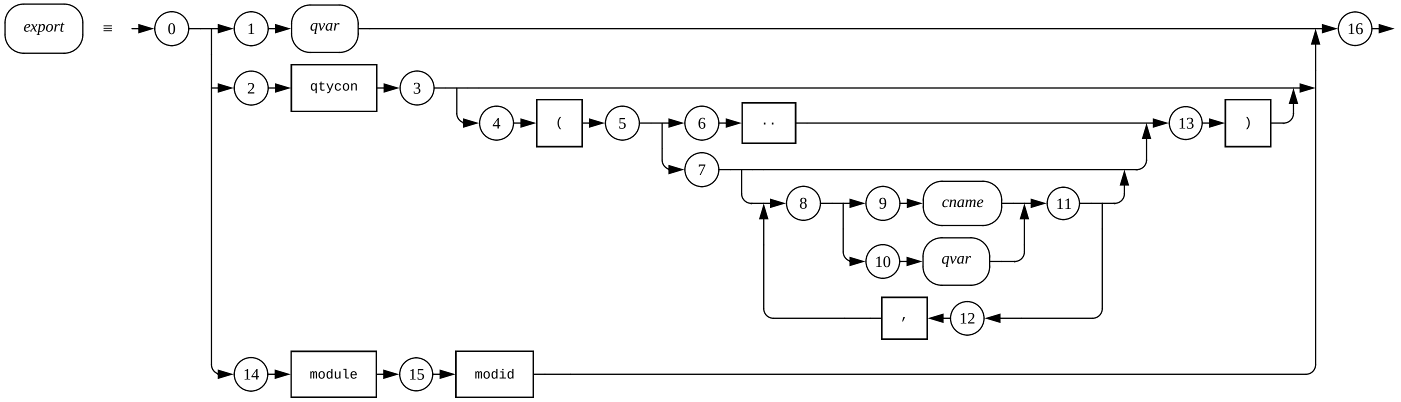
[30] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

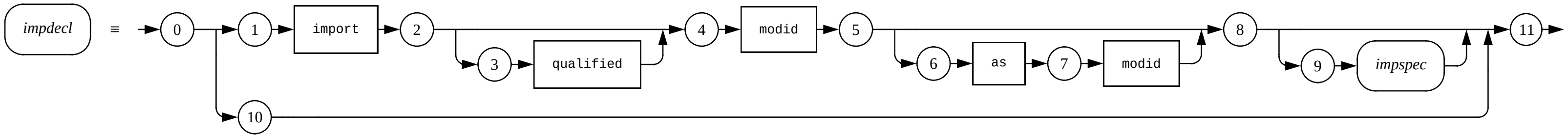
[31] P. Zadarnowski. *λTEX*. Available from URL: `http://www.jantar.org/lambdaTeX/`.

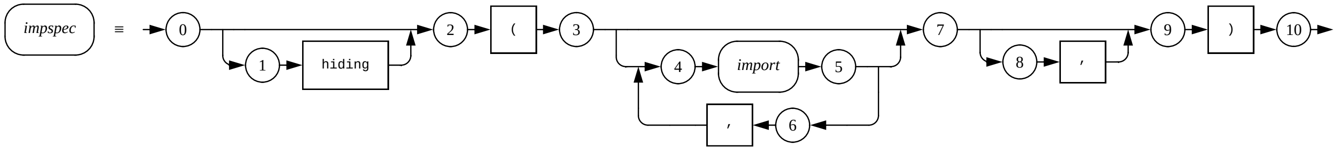
A Automaton description

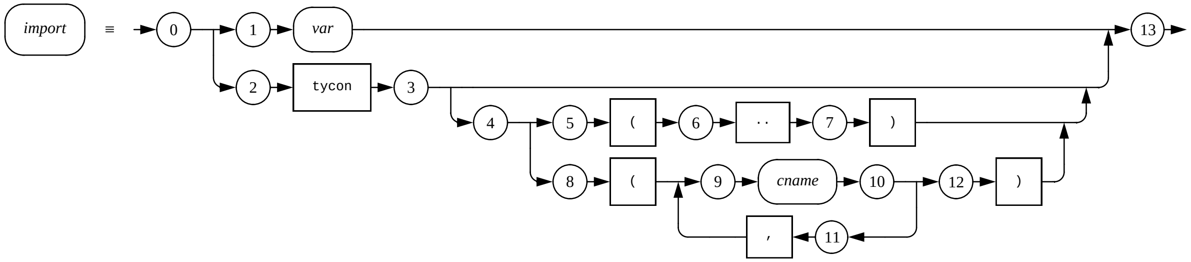
These diagrams show the relationship between the lookup table of the automaton and the Haskell context free grammar. The same program, which is based on the PC library we have developed, is used to generate both Lout [11] mark-up files, which Lout then transforms into these diagrams, and the executable code. Another program, also based on our PC library, parses the EPS files created by Lout and removes unused functions, comments and blank lines; this represents a space saving of 50%, or approximately 5.5M on the uncompressed postscript document and 800k on the gzipped copy.

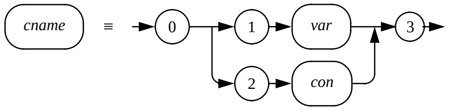


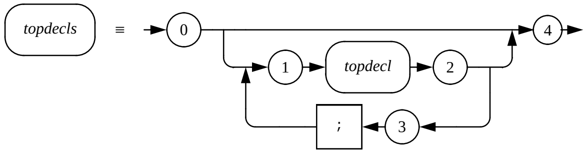

export
≡
0
1
qvar
16
2
qtycon
3
4
(
5
6
..
13
)
7
8
9
cname
11
10
qvar
,
12
14
module
15
modid

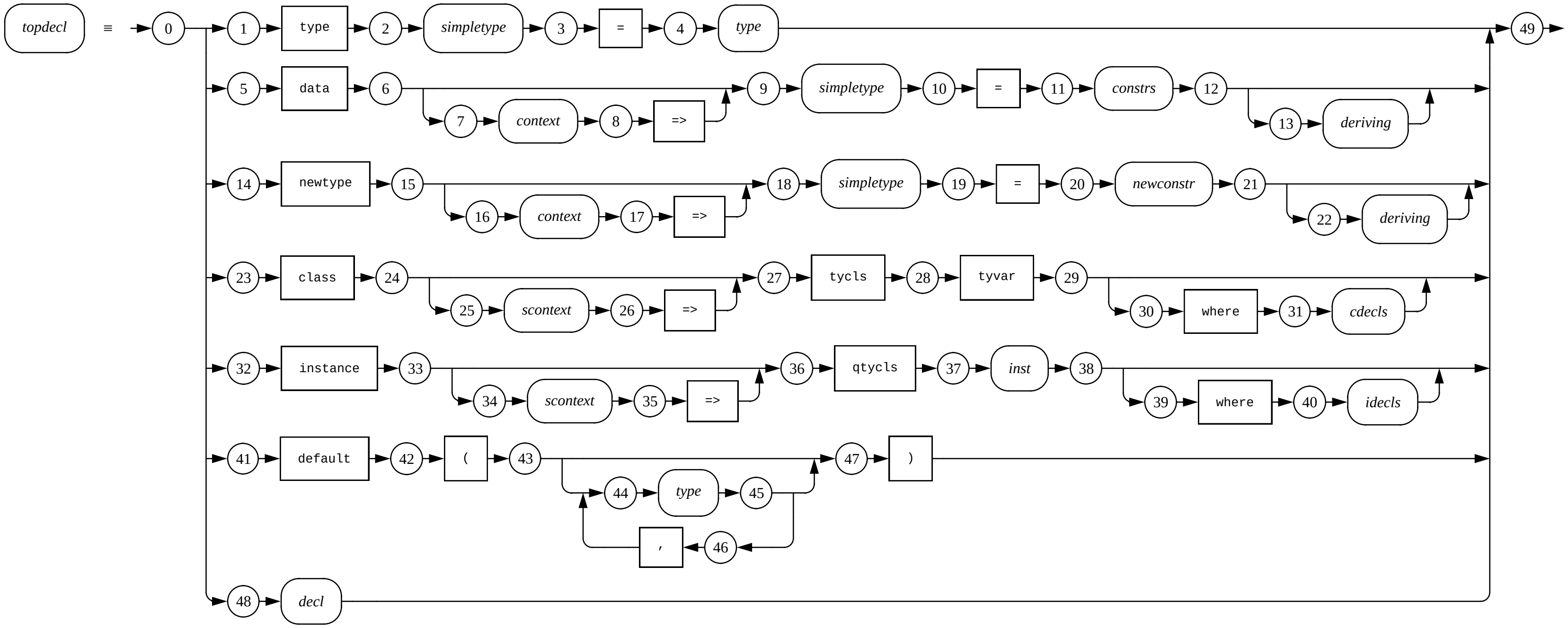

impdecl
≡
0
1
import
2
3
qualified
4
modid
5
6
as
7
modid
8
9
impspec
11
10

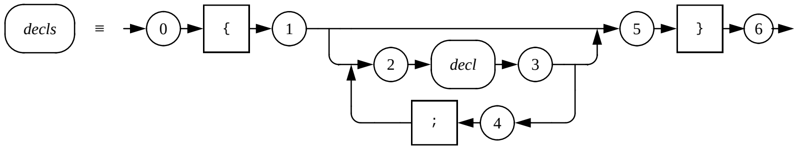

impspec
≡
0
1
hiding
2
(
3
4
import
5
,
6
7
8
,
9
)
10

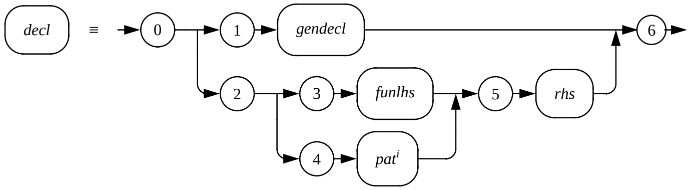

import
≡
0
1
var
13
2
tycon
3
4
5
(
6
..
7
)
8
(
9
cname
10
12
)
,
11

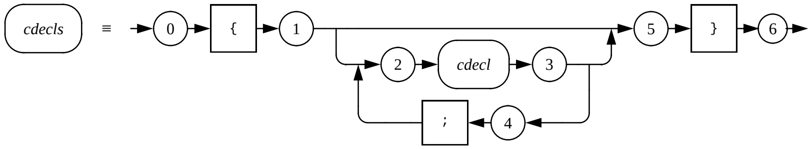

cname
≡
0
1
var
3
2
con

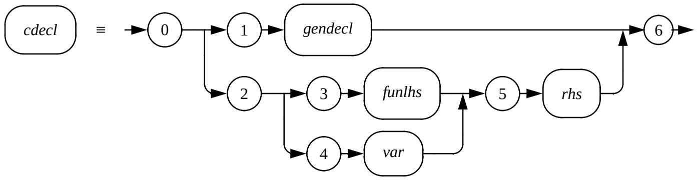

topdecls
≡
0
4
1
topdecl
2
;
3

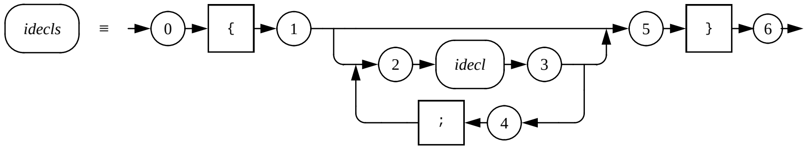

topdecl
≡
type
simpletype
=
type
data
context
=>
simpletype
=
constrs
deriving
newtype
context
=>
simpletype
=
newconstr
deriving
class
scontext
=>
tycls
tyvar
where
cdecls
instance
scontext
=>
qtycls
inst
where
idecls
default
(
type
,
)
decl

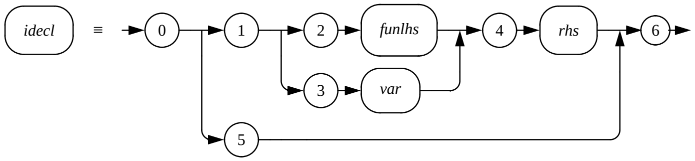

decls
≡
{
decl
;
}

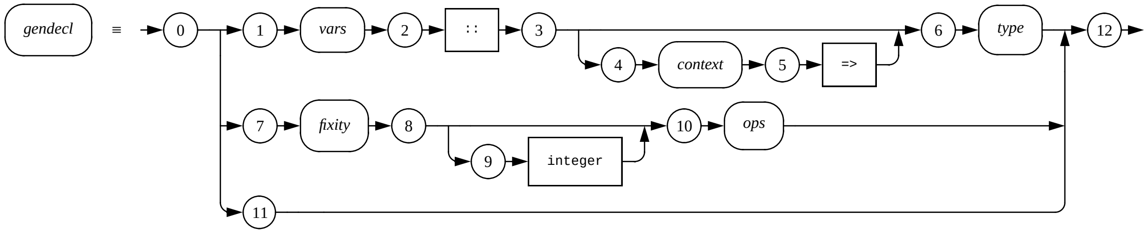

decl
≡
0
1
gendecl
6
2
3
funlhs
5
rhs
4
pat^{i}

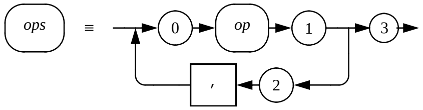

cdecls
≡
0
{
1
5
}
6
2
cdecl
3
;
4

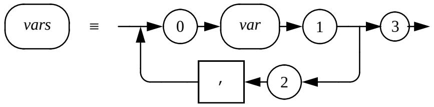

cdecl
≡
0
1
gendecl
6
2
3
funlhs
5
rhs
4
var

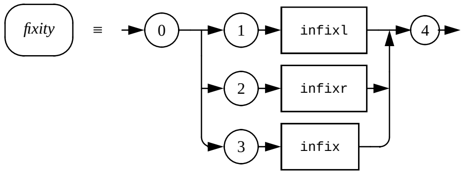

idecls
≡
0
{
1
5
}
6
2
idecl
3
;
4

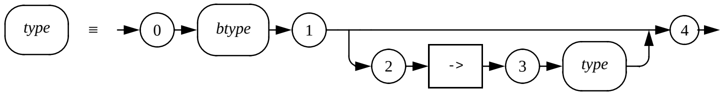

idecl
≡
0
1
2
funlhs
3
var
4
rhs
5
6

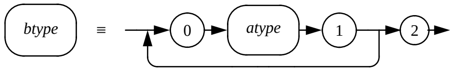

gendecl
≡
0
1
vars
2
::
3
4
context
5
=>
6
type
12
7
fixity
8
9
integer
10
ops
11

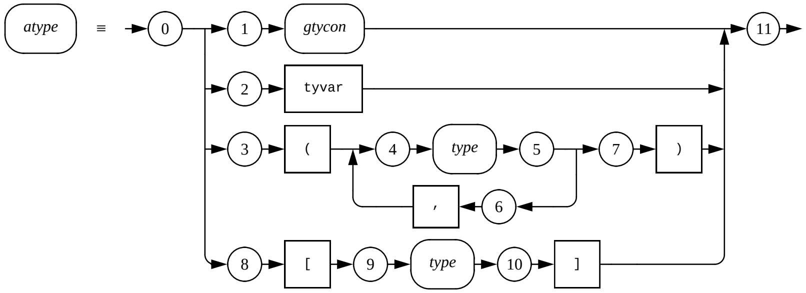

ops
≡
0
op
1
3
,
2

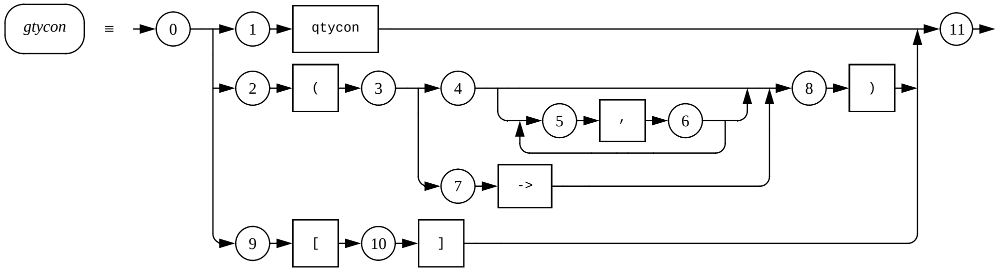

vars
≡
0
var
1
3
,
2

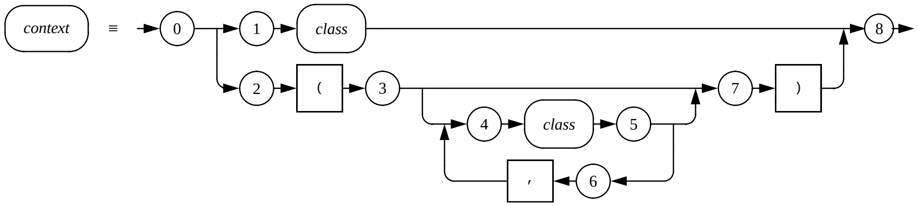

fixity
≡
0
1
infixl
4
2
infixr
3
infix

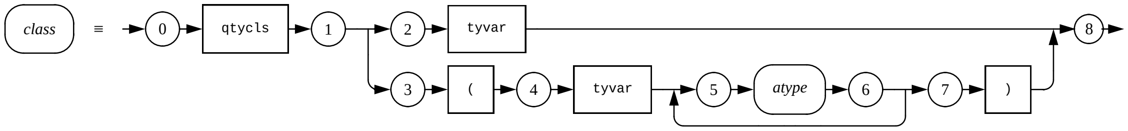

type
≡
0
btype
1
2
->
3
type
4

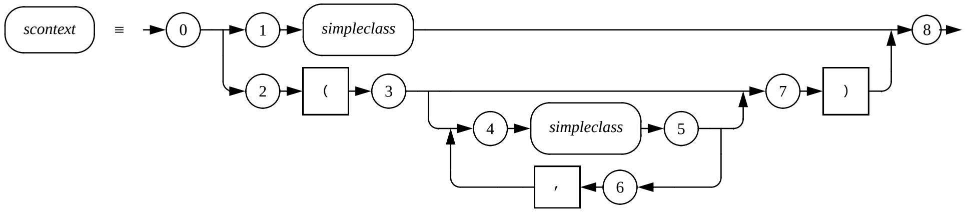

btype
≡
0
atype
1
2

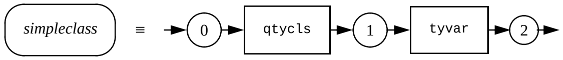

atype
≡
0
1
gtycon
11
2
tyvar
3
(
4
type
5
7
)
,
6
8
[
9
type
10
]

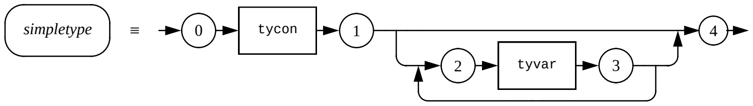

gtycon
≡
0
1
qtycon
11
2
(
3
4
8
)
5
,
6
7
->
9
[
10
]

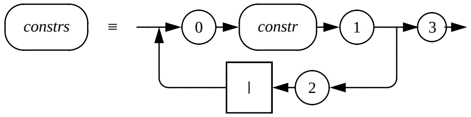

context
≡
0
1
class
8
2
(
3
4
class
5
,
6
7
)

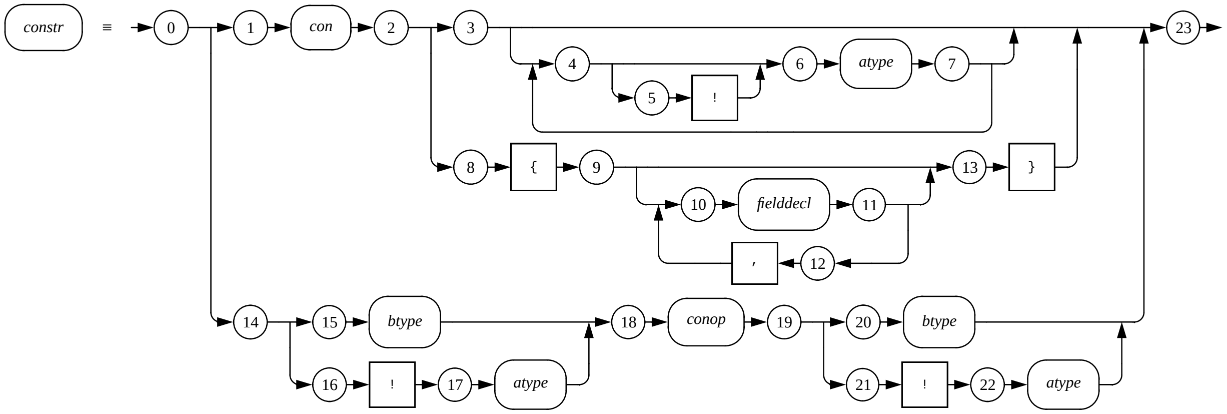

class
≡
0
qtycls
1
2
tyvar
8
3
(
4
tyvar
5
atype
6
7
)

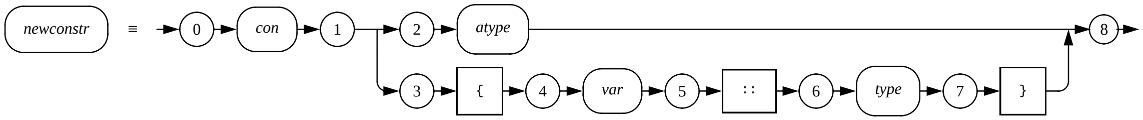

scontext
≡
0
1
simpleclass
8
2
(
3
4
simpleclass
5
,
6
7
)

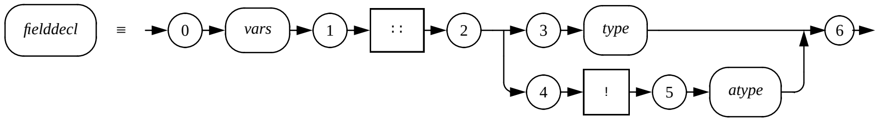

simpleclass
≡
0
qtycls
1
tyvar
2

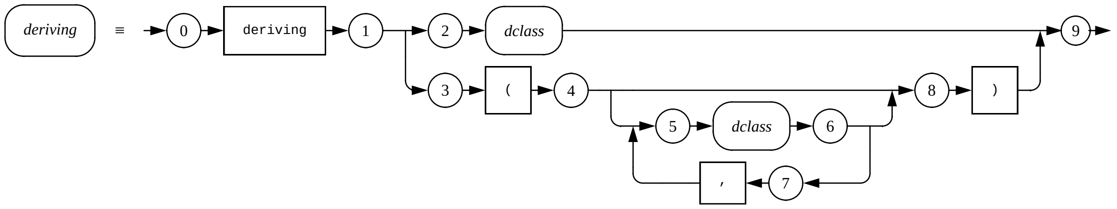

simpletype
≡
0
tycon
1
2
tyvar
3
4

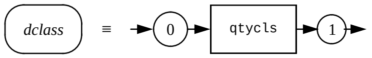

constrs
≡
0
constr
1
3
|
2

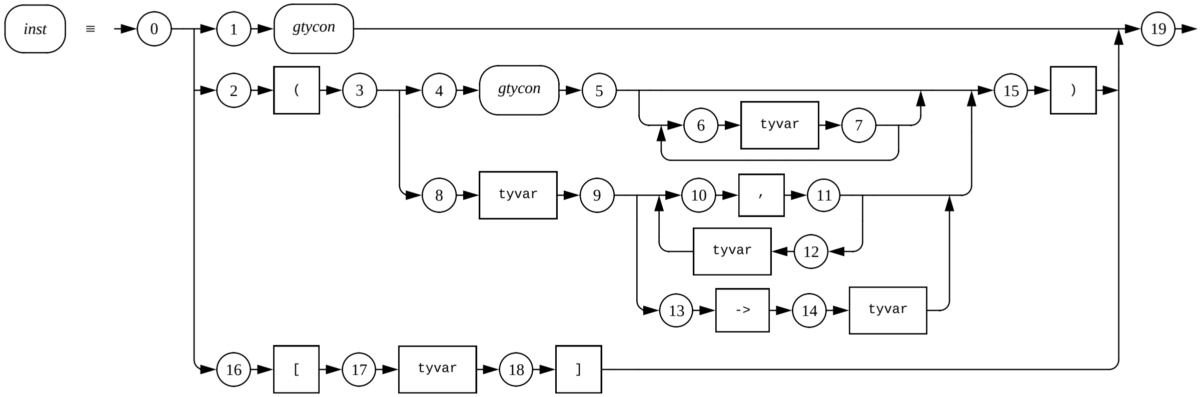

constr
≡
0
1
con
2
3
23
4
6
atype
7
5
!
8
{
9
13
}
10
fielddecl
11
,
12
14
15
btype
18
conop
19
20
btype
16
!
17
atype
21
!
22
atype

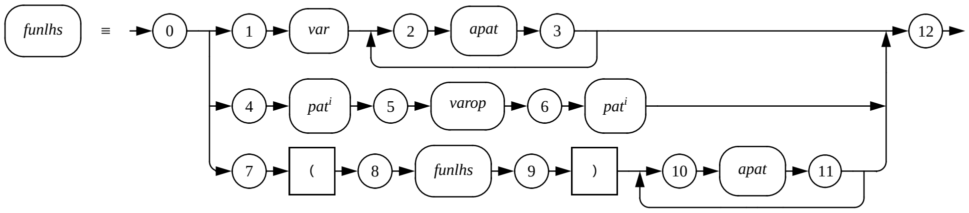

newconstr
≡
0
con
1
2
atype
8
3
{
4
var
5
::
6
type
7
}


fielddecl
≡
0
vars
1
::
2
3
type
6
4
!
5
atype

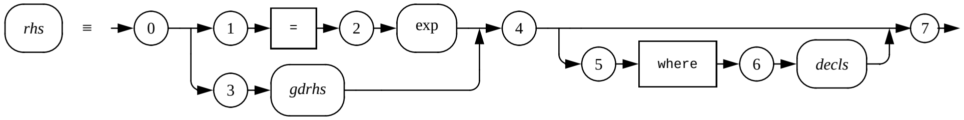

deriving
≡
0
deriving
1
2
dclass
9
3
(
4
8
)
5
dclass
6
,
7


dclass
≡
0
qtycls
1

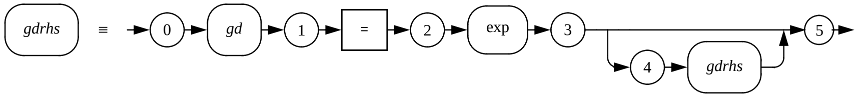

inst
≡
0
1
gtycon
19
2
(
3
4
gtycon
5
15
)
6
tyvar
7
8
tyvar
9
10
,
11
tyvar
12
13
->
14
tyvar
16
[
17
tyvar
18
]



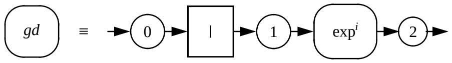
funlhs
≡
0
1
var
2
apat
3
12
4
pat^i
5
varop
6
pat^i
7
(
8
funlhs
9
)
10
apat
11



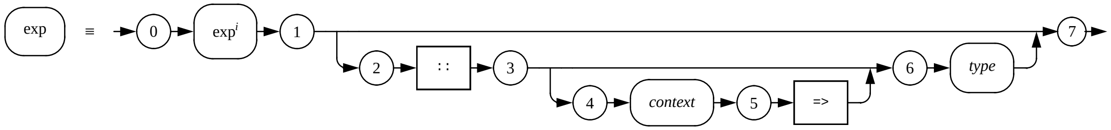
rhs
≡
0
1
=
2
exp
4
7
3
gdrhs
5
where
6
decls



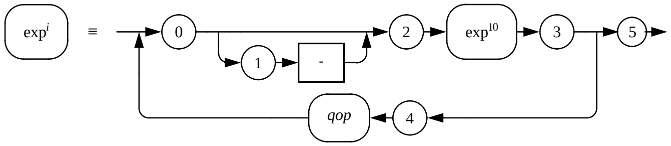
gdrhs
≡
0
gd
1
=
2
exp
3
5
4
gdrhs



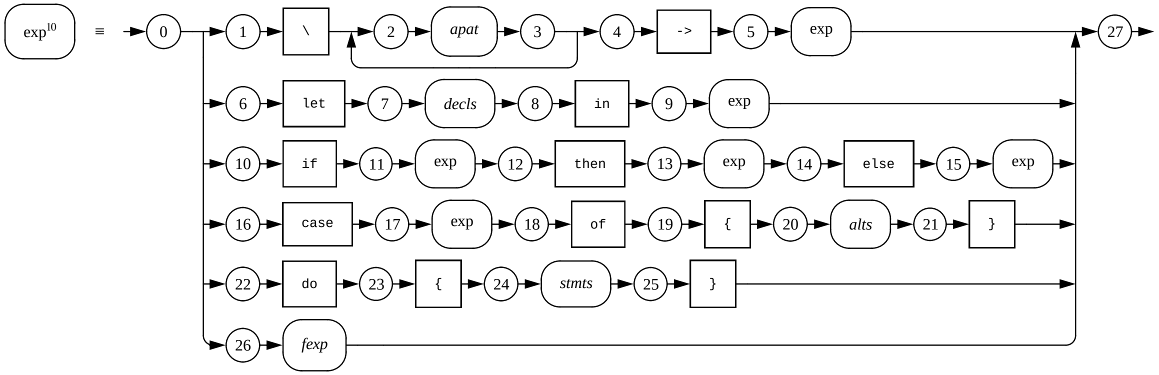
gd
≡
0
|
1
exp^i
2

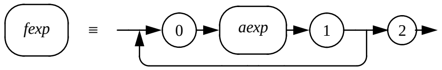


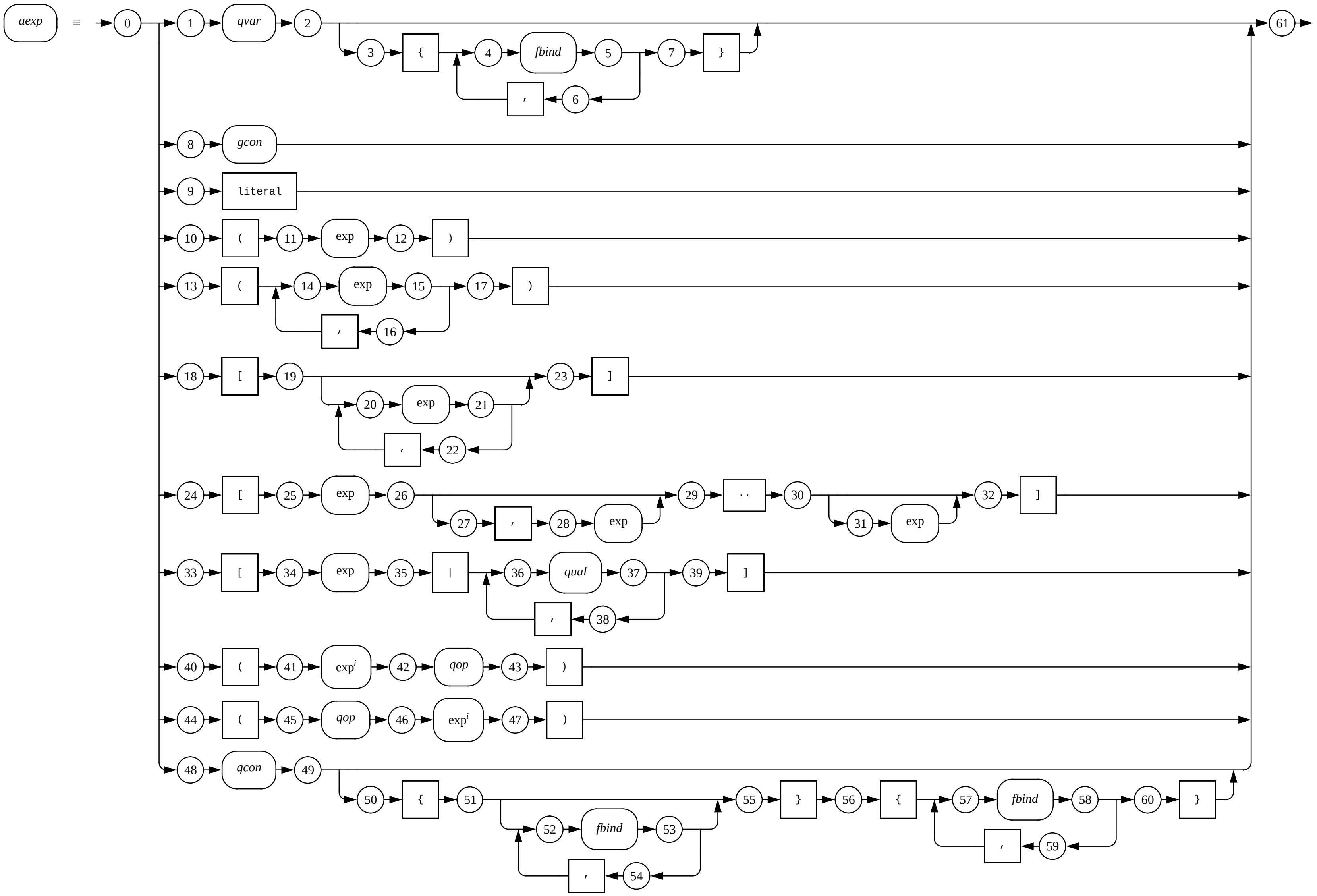
exp
≡
0
exp^i
1
7
2
: :
3
6
type
4
context
5
=>

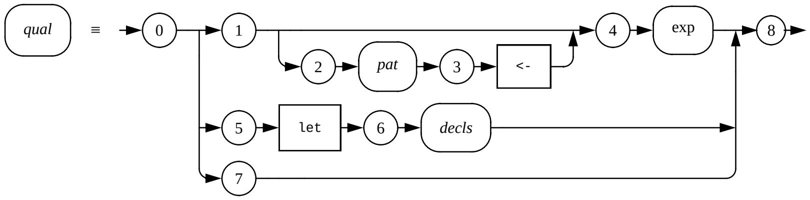


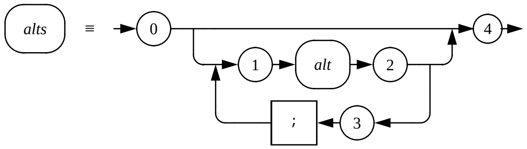
exp^i
≡
0
2
exp^{10}
3
5
1
-
qop
4

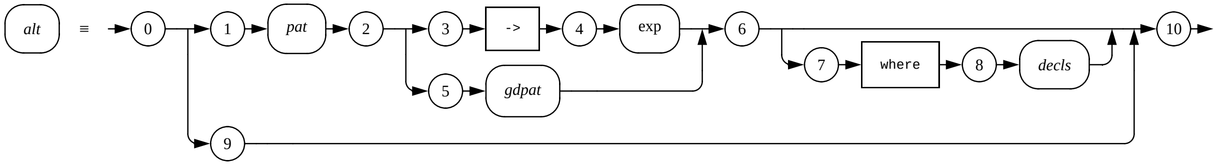

exp^10
≡
0
1
\
2
apat
3
4
->
5
exp
27
6
let
7
decls
8
in
9
exp
10
if
11
exp
12
then
13
exp
14
else
15
exp
16
case
17
exp
18
of
19
{
20
alts
21
}
22
do
23
{
24
stmts
25
}
26
fexp

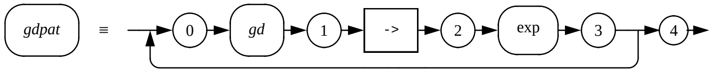

fexp
≡
0
aexp
1
2

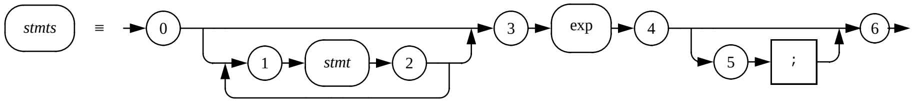

aexp
≡
0
1
qvar
2
61
3
{
4
fbind
5
7
}
,
6
8
gcon
9
literal
10
(
11
exp
12
)
13
(
14
exp
15
17
)
,
16
18
[
19
23
]
20
exp
21
,
22
24
[
25
exp
26
29
· ·
30
32
]
27
,
28
exp
31
exp
33
[
34
exp
35
|
36
qual
37
39
]
,
38
40
(
41
exp^i
42
qop
43
)
44
(
45
qop
46
exp^i
47
)
48
qcon
49
50
{
51
55
}
56
{
57
fbind
58
60
}
52
fbind
53
,
54
,
59

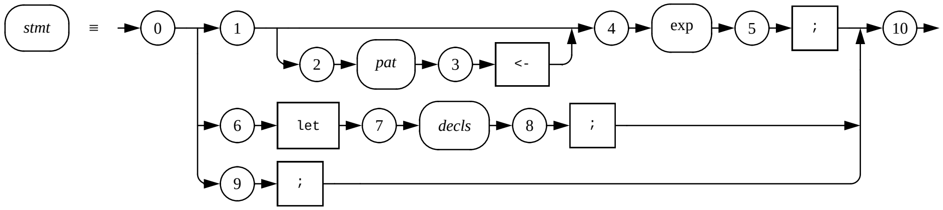

qual
≡
0
1
4
exp
8
2
pat
3
<-
5
let
6
decls
7

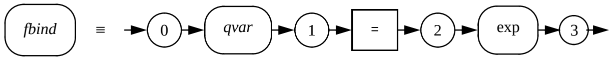

alts
≡
0
4
1
alt
2
;
3

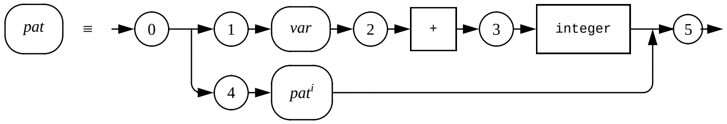

alt
≡
0
1
pat
2
3
->
4
exp
6
10
7
where
8
decls
5
gdpat
9

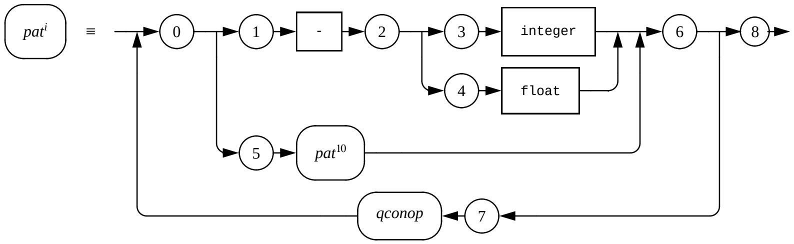

gdpat
≡
0
gd
1
->
2
exp
3
4

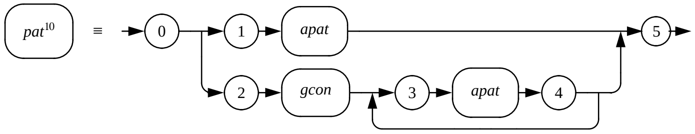

stmts
≡
0
3
exp
4
6
1
stmt
2
5
;

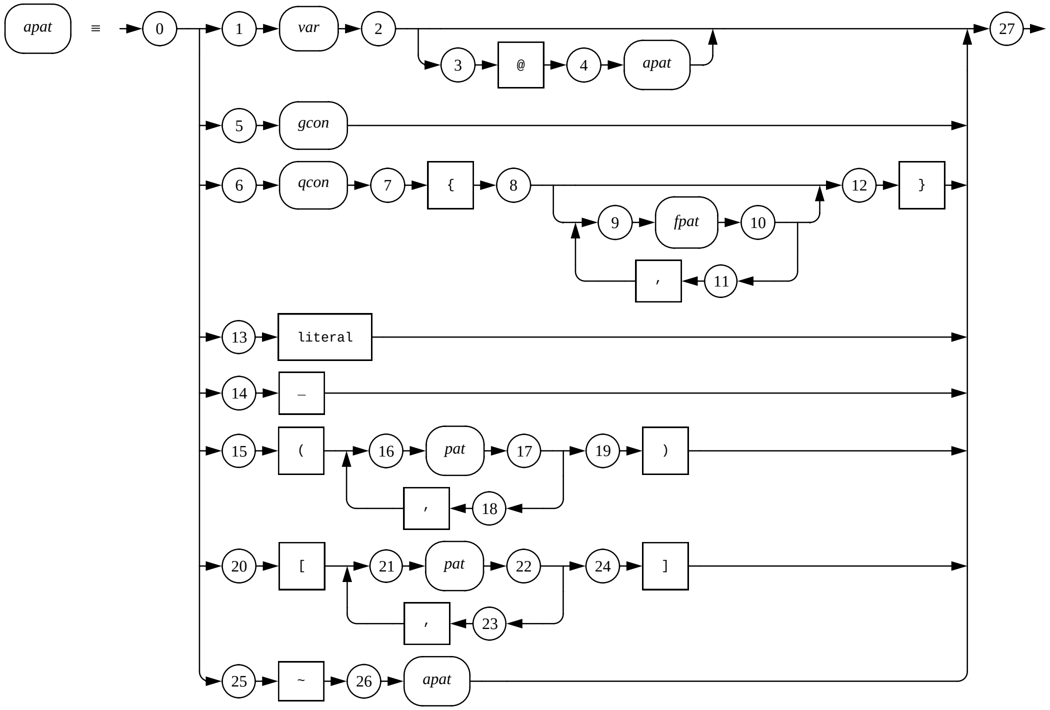

stmt
≡
0
1
2
pat
3
<-
4
exp
5
;
10
6
let
7
decls
8
;
9
;

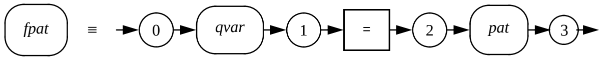

fbind
≡
0
qvar
1
=
2
exp
3

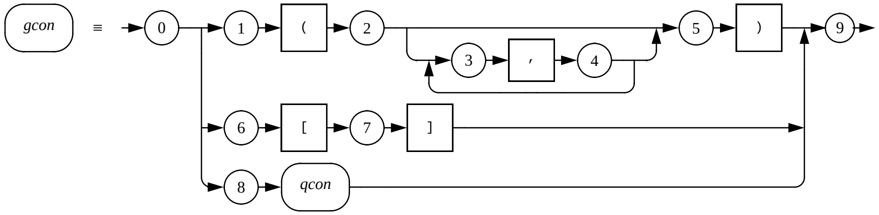

pat
≡
0
1
var
2
+
3
integer
5
4
pat^i

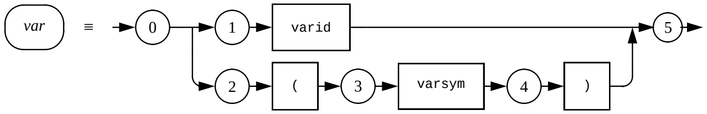

pat^i
≡
0
1
-
2
3
integer
4
float
6
8
5
pat^{10}
qconop
7

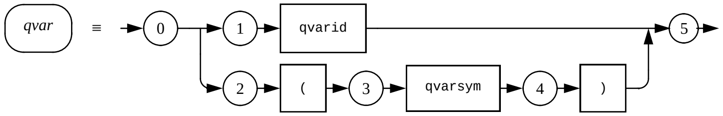

pat^{10}
≡
0
1
apat
5
2
gcon
3
apat
4

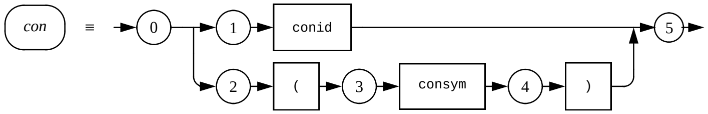

apat
≡
0
1
var
2
3
@
4
apat
27
5
gcon
6
qcon
7
{
8
9
fpat
10
,
11
12
}
13
literal
14
_
15
(
16
pat
17
19
)
,
18
20
[
21
pat
22
24
]
,
23
25
~
26
apat

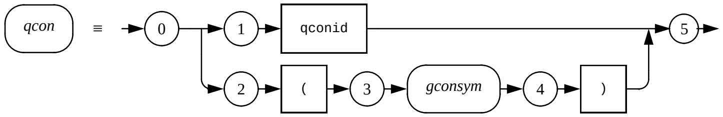

fpat
≡
0
qvar
1
=
2
pat
3

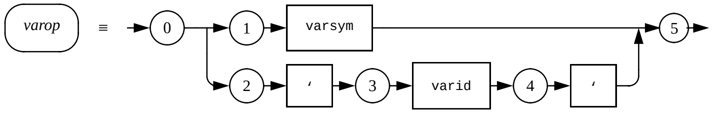

gcon
≡
0
1
(
2
3
,
4
5
)
9
6
[
7
]
8
qcon

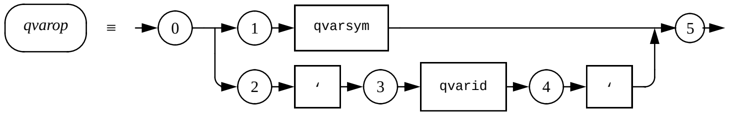

var
≡
0
1
varid
5
2
(
3
varsym
4
)

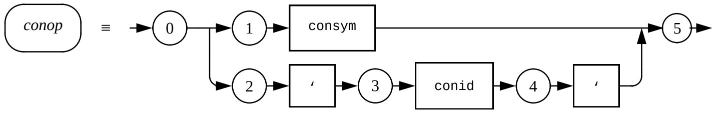

qvar
≡
0
1
qvarid
5
2
(
3
qvarsym
4
)

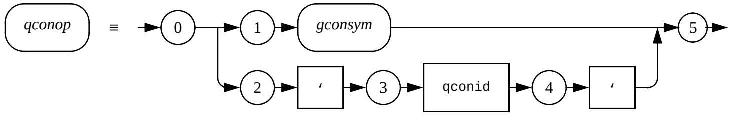

con
≡
0
1
conid
5
2
(
3
consym
4
)

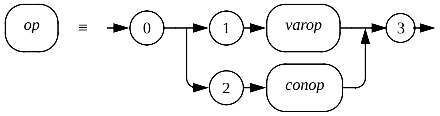

qcon
≡
0
1
qconid
5
2
(
3
gconsym
4
)

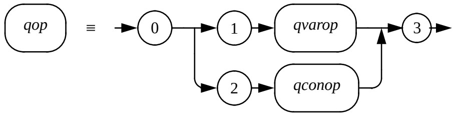

varop
≡
0
1
varsym
5
2
‘
3
varid
4
‘

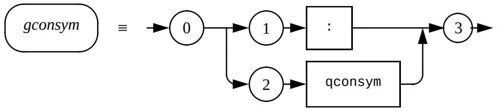

qvarop
≡
0
1
qvarsym
5
2
‘
3
qvarid
4
‘


conop
≡
0
1
consym
5
2
‘
3
conid
4
‘


qconop
≡
0
1
gconsym
5
2
‘
3
qconid
4
‘


op
≡
0
1
varop
3
2
conop


qop
≡
0
1
qvarop
3
2
qconop


gconsym
≡
0
1
:
2
qconsym
3