CHALMERS

Robust & Precise incremental parsing of Haskell

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and logic

ANDERS KARLSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, August 2009

Robust & Precise incremental parsing of Haskell

Anders Karlsson

Supervisor: Jean-Philippe Bernardy
Examiner: Koen Claessen

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2009

Abstract

By using an incremental parser library, this thesis will implement a parser for the functional programming language Haskell. The parser will be used in Yi, a text editor that allows precise editing modes using an abstract syntax tree provided by the incremental parser library.

We will explain how to find errors in the code and how to mark them so that they can be visually marked to the user in Yi. The parser is intended to give feedback online, i.e. it will parse at each keystroke so that errors are marked as soon as they appear. Problems that have occured during the development and the solution to them will be described.

Sammanfattning

Med hjälp av ett inkrementell parserbibliotek, kommer vi i denna tes att implementera en parser för det funktionella programmeringsspråket Haskell. Parsern kommer att användas i Yi, en textredigerare som tillåter noggranna redigeringslägen med hjälp av ett abstrakt syntaxträd, tillhandahållet av det inkrementella parserbiblioteket.

Det kommer förklaras hur man hittar fel i koden och hur man markerar dessa så att de visuellt kan markeras för användaren i Yi. Parsern skall kunna ge respons “online”, dvs. den skall ge respons vid varje knapptryckning så att fel är markerade så snart de uppenbarar sig. Problem som stötts på under utvecklingen och deras lösningar kommer beskrivas.

Acknowledgements

First of all I would like to thank my supervisor Jean-Philippe Bernardy for his guidance and support during this project.

I would also like to thank Deniz Dogan for our many interesting discussions about the parser library, Yi in general, modes in Yi and Haskell. Further kudos, in no specific order, goes to Andreas Svensson, Tobias Olausson, Koen Claessen and Jeff Wheeler.

Thanks to Fredrik Andersson for being my opponent for this thesis.

Contents

List of Figures

1 Introduction

1.1 Motivation

Haskell is a functional programming language that makes use of lazy evaluation (call-by-need). It supports higher-order functions and currying. Haskell code is easy to structure well and the code can often be reused due to the support of higher-order functions and laziness (Hughes, 1989).

Today many programming environments that support Haskell development, provide user feedback interactively. The most common problems with such environments is that they

- Use too much memory and/or CPU time due to the complex parsers and compilers being used. Also they often re-parse everything instead of reusing previous information.
- Spend time on parsing code that is not necessary to parse, re-parsing things that have not been changed should not be necessary, instead the earlier abstract syntax tree should be reused as much as possible.
- Use regular-expression based parsers that can not recognize complex structures, this does not give as precise information as wanted.
- Compile parts or all of the code which give very precise information, but with the drawback that the feedback is usually delayed since compiling can take some extra time. Also the the parser in the compiler is not robust enough and often does not recover from errors.

These problems force the user to choose between precise feedback, but with a delay and probably only one of many errors will be found at first run or imprecise information available after each keystroke.

2 Aim

Our parser should give feedback to the user by combining the strengths of the existing solutions, but without their drawbacks, it should provide:

Feedback online the feedback should be available immediately, at each keystroke.

Precise feedback the parser should be able to recognize complex structures.

Robustness the parser should cope with errors and recover from them.

Encapsulation the errors should be encapsulated so that everything after an error can be highlighted correctly.

The technical goals of the parser in this project are

Parse Haskell98 being able to parse most of the Haskell98 syntax.

Incremental parser the existing incremental parser library that is a part of Yi should be used.

Available in Yi the parser should be available as a mode in Yi (sec.3.3).

3 Background

3.1 Haskell

Even though the reader of this thesis is expected to have basic knowledge of Haskell, details of the syntax are important in this thesis. Therefore this part of the thesis will be a short introduction to the parts of Haskell that will be parsed in this thesis. The information is collected from the Haskell98 revised report (Peyton Jones, 2003).

The most widely used compiler, the Glasgow Haskell Compiler (of Glasgow, n.d.), will be used in this project. Haskell is based on the Haskell98 report (Peyton Jones, 2003) and since then several extensions has been proposed and implemented. We aim to parse Haskell98 with some of the extensions provided by GHC.

The notational conventions, used in the Haskell98 report (fig. 1) will be used in this report to describe the syntax of Haskell.

$[\mathit{pattern}]$	*optional*
$\{\mathit{pattern}\}$	*zero or more repetitions*
$(\mathit{pattern})$	*grouping*
$\mathit{pat1} \mid \mathit{pat2}$	*choice*
$\mathit{pat} < \mathit{pat}' >$	*difference* -- elements generated by pat *except those generated by pat′*

Figure 1: Notational conventions

3.1.1 Identifiers and symbols

Variable and constructor identifiers (fig. 2) are distinguished by their first character. If the first character is capitalized, then the token is a constructor identifier and if not then it is a variable identifier.

$$\mathit{varid} \rightarrow (\mathit{small}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid {}'\}) < \mathit{reservedid} >$$
$$\mathit{conid} \rightarrow \mathit{large}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid {}'\}$$

Figure 2: Variable and constructor identifiers

In this thesis qualified variable and constructor identifiers will be treated as usual variable and constructor identifiers, instead of as in the haskell98 report, which treats them separately.

$$
\begin{array}{lll}
\mathit{symbol} & \rightarrow & \mathit{ascSymbol} \mid \mathit{uniSymbol} < \mathit{special} \mid \_ \mid : \mid \text{"} \mid {}' > \\
\mathit{ascSymbol} & \rightarrow & ! \mid \# \mid \$ \mid \% \mid \& \mid * \mid + \mid \circ \mid / \mid < \mid = \mid > \mid ? \mid @ \\
 & & \mid \backslash \mid \uparrow \mid\mid\mid - \mid \sim \\
\mathit{special} & \rightarrow & (\mid) \mid , \mid ; \mid [\mid] \mid \text{`} \mid \{ \mid \} \\
\mathit{uniSymbol} & \rightarrow & \mathit{any\ Unicode\ symbol\ or\ punctuation} \\
\mathit{varsym} & \rightarrow & (\mathit{symbol}\ \{\mathit{symbol} \mid :\}) < \mathit{reservedop} \mid \mathit{dashes} > \\
\mathit{consym} & \rightarrow & (:\{\mathit{symbol} \mid :\}) < \mathit{reservedop} > \\
\mathit{reservedop} & \rightarrow & .. \mid : \mid :: \mid = \mid \lambda \mid\mid\mid \leftarrow \mid \rightarrow \mid @ \mid \sim \mid \Rightarrow
\end{array}
$$

Figure 3: Symbol identifiers

Infix functions (‘*function*‘) are treated as *varsym* and infix data constructors are treated as *consym* in this thesis.

3.1.2 Comments

A comment in Haskell can be placed anywhere and has the following structure

$$
\begin{array}{lll}
\mathit{comment} & \rightarrow & \mathit{dashes}\ [\mathit{any} < \mathit{symbol} > \{\mathit{any}\}]\ \mathit{newline} \\
\mathit{dashes} & \rightarrow & \texttt{--}\ \{\texttt{-}\} \\
\mathit{opencom} & \rightarrow & \{- \\
\mathit{closecom} & \rightarrow & -\}
\end{array}
$$

3.1.3 Reserved words

Haskell98 only has 21 keywords[1] which are not allowed as names for values or types (Hudak, Hughes, Jones and Wadler, 2007). But it has an exception, three of the keywords is not reserved. These quasi-keywords can be used as regular identifiers. A list of the quasi-keywords can be viewed in fig. 4.

$$\mathit{as} \mid \mathit{qualified} \mid \mathit{hiding}$$

Figure 4: Keywords that is not reserved

These quasi-keywords pose an interesting problem: they should be highlighted as keywords when used as such and as ordinary Identifiers when not.

[1]Compared to Javas 50 keywords

We will show how to incrementally parse these quasi-keywords and then highlight them appropriately in section 4.4.6.

The reserved words in Haskell are listed in fig. 5.

$$
\begin{aligned}
\textit{reservedid} \rightarrow\ & \textbf{case} \mid \textbf{class} \mid \textbf{data} \mid \textbf{default} \mid \textbf{deriving} \mid \textbf{do} \mid \textbf{else} \\
& \mid \textbf{if} \mid \textbf{import} \mid \textbf{in} \mid \textbf{infix} \mid \textbf{infixl} \mid \textbf{infixr} \\
& \mid \textbf{instance} \mid \textbf{let} \mid \textbf{module} \mid \textbf{newtype} \mid \textbf{of} \mid \textbf{then} \\
& \mid \textbf{type} \mid \textbf{where} \mid \_
\end{aligned}
$$

Figure 5: Reserved keywords in Haskell98

3.1.4 Module declaration

In a module, only comments and pragmas are allowed above the module declaration. The module declaration specifies what the module should export, everything else will be internal only. The module declaration is optional and if not declared, everything in the module is exported. The structure of a module declaration can be seen in fig. 6.

$$
\begin{array}{lll}
\textit{moddecl} & \rightarrow \textbf{module}\ \textit{conid}\ [\textit{exports}]\ \textbf{where} & \\
\textit{exports} & \rightarrow (\textit{export1}, ..., \textit{exportn}\ [,]) & (n \geqslant 0) \\
\textit{export} & \rightarrow \textit{var} & \\
 & \quad \mid \textit{conid}\ [(..) \mid (\textit{cname1}, ..., \textit{cnamen})] & (n \geqslant 0) \\
 & \quad \mid \textit{conid}\ [(..) \mid (\textit{var1}, ..., \textit{varn})] & (n \geqslant 0) \\
 & \quad \mid \textbf{module}\ \textit{conid} & \\
\textit{cname} & \rightarrow \textit{var} \mid \textit{con} & \\
\textit{var} & \rightarrow \textit{varid} \mid (\textit{varsym}) & \\
\textit{con} & \rightarrow \textit{conid} \mid (\textit{consym}) &
\end{array}
$$

Figure 6: Module declaration

The *conid* after the **module** keyword, represents the name of the module fig. 6. The exports, which are optional, are separated by commas and has an optional trailing comma. If no exports are given, everything will be exported and if the export field is empty, nothing is exported.

3.1.5 Imports

The imports follow directly after the module declaration if there is one, otherwise imports must be located in the beginning of the module.

An example of the import structure can be seen in fig. 7, a *qualified* import is hidden but its exported content can instead be reached by qualifying. The *as*

$$
\begin{array}{lll}
\textit{impdecl} & \rightarrow \textbf{import}\ [\textit{qualified}]\ \textit{conid}\ [\textit{as conid}]\ [\textit{impspec}] & \\
\textit{impspec} & \rightarrow (\textit{import1}, ..., \textit{importn}\ [,]) & (n \geqslant 0) \\
 & \mid \textit{hiding}\ (\textit{import1}, ..., \textit{importn}\ [,]) & (n \geqslant 0) \\
\textbf{import} & \rightarrow \textit{var} & \\
 & \mid \textit{tycon}\ [(..) \mid (\textit{cname1}, ..., \textit{cnamen})] & (n \geqslant 0) \\
 & \mid \textit{tycls}\ [(..) \mid (\textit{var1}, ..., \textit{varn})] & (n \geqslant 0) \\
\textit{cname} & \rightarrow \textit{var} \mid \textit{con} & \\
\textit{var} & \rightarrow \textit{varid} \mid (\textit{varsym}) & \\
\textit{con} & \rightarrow \textit{conid} \mid (\textit{consym}) &
\end{array}
$$

Figure 7: Import declaration

quasi-keyword used in imports changes the name locally of the imported module. The *hiding* quasi-keyword is used to hide some of the exported content of the imported module. The import list at the end specifies what to import or, if the *hiding* quasi-keyword is used not to import. The import list is separated by commas, with an optional ending comma. If no import list is given, everything will be imported, and if the list is empty, nothing will be imported.

3.1.6 Declarations (top)

A data type can contain data constructors, which like type constructors is identifiers with a beginning capitalized character, type constructors and type variables. fig. 8 describes how a data type can be declared. The right hand side (rhs) begins after the equal sign.

newtype allows wrapping a type synonym in a constructor so it can be matched by pattern matching. It does not add any overhead to the run time, like **data**, since it will be replaced at compile-time. The syntax used can be seen in fig. 9.

A type synonym can be declared like in fig. 10. The first *conid* will be the name of the type synonym.

3.1.7 Expressions

Haskell has several different expressions, but in this thesis only **let** expressions has been considered important to parse specially, this will be further described in section 4.4.15. A let expressions allows the user to declare several declarations, ending with an expression. In fig. 11, for example the two declarations $a = 2 * 2$ and $b = 4 * 4$ ends in the **in** part which is an expression, in this case $a * b$ so the result of this calculation is 64. The ending **in** is optional inside of a **do** block.

$$\begin{array}{lll}
\textit{datadecl} & \rightarrow & \textbf{data}\ [\textit{context} \Rightarrow]\ \textit{simpletype} = \textit{constrs}\ [\textbf{deriving}\ ] \\
\textit{simpletype} & \rightarrow & \textit{conid varid1 ... varidk} \qquad (k \geqslant 0) \\
\textit{constrs} & \rightarrow & \textit{constr1} \mid ... \mid \textit{constrn} \qquad (n \geqslant 1) \\
\textit{constr} & \rightarrow & \textit{con}\ [!]\ \textit{atype1}\ ...\ [!]\ \textit{atypek}\ (\textit{arity con} = k, k \geqslant 0) \\
 & & \mid (\textit{btype} \mid !\textit{atype})\ \textit{consym}\ (\textit{btype} \mid !\textit{atype})\ (\textbf{infix}\ \textit{consym}) \\
 & & \mid \textit{con}\ \{\textit{fielddecl1}, ..., \textit{fielddecln}\} \qquad (n \geqslant 0) \\
\textit{fielddecl} & \rightarrow & \textit{vars} :: (\textbf{type} \mid !\textit{atype}) \\
\textbf{deriving} & \rightarrow & \textbf{deriving}\ (\textit{conid} \mid (\textit{conid1}, ..., \textit{conidn})) \quad (n \geqslant 0) \\
\textbf{type} & \rightarrow & \textit{btype}\ [\rightarrow \textbf{type}\ ] \quad (\textit{function}\ \textbf{type}\ ) \\
\textit{btype} & \rightarrow & [\textit{btype}]\ \textit{atype} \qquad (\textbf{type}\ \textit{application}) \\
\textit{atype} & \rightarrow & \textit{gtycon} \\
 & & \mid \textit{varid} \\
 & & \mid (\textit{type1}, ..., \textit{typek})\ (\textit{tuple}\ \textbf{type}\ , k \geqslant 2) \\
 & & \mid [\textbf{type}\ ] \qquad (\textit{list}\ \textbf{type}\ ) \\
 & & \mid (\textbf{type}\ ) \qquad (\textit{parenthesised constructor}) \\
\textit{gtycon} & \rightarrow & \textit{conid} \\
 & & \mid () \qquad (\textit{unit}\ \textbf{type}\ ) \\
 & & \mid [\,] \qquad (\textit{list constructor}) \\
 & & \mid (\rightarrow) \qquad (\textit{function constructor}) \\
 & & \mid (, \{, \}) \ (\textit{tupling constructors})
\end{array}$$

Figure 8: Data declaration

$$\begin{array}{lll}
\textit{ntypdecl} & \rightarrow & \textbf{newtype}\ [\textit{context} \Rightarrow]\ \textit{simpletype} = \textit{newconstr}\ [\textbf{deriving}\ ] \\
\textit{newconstr} & \rightarrow & \textit{con atype} \\
 & & \mid \textit{con}\ \{\ \textit{var} :: \textbf{type}\ \} \\
\textit{simpletype} & \rightarrow & \textit{conid varid1 ... varidk}\ (k \geqslant 0)
\end{array}$$

Figure 9: Newtype declaration

$$\textit{typedecl} \rightarrow \textbf{type}\ \textit{simpletype} = \textbf{type}$$

Figure 10: Type declaration

3.2 Layout

In order to separate sequential commands most languages use semicolon (Hudak et al., 2007), Haskell does not have sequential commands but there is a need to separate declarations. The declarations are usually separated by the layout which makes the syntax a bit less cluttered. In Haskell the indentation require-

```
let a = 2 * 2
    b = 4 * 4
in  a * b
```

Figure 11: Simple let expression

ment can be replaced, by explicit structuring, using ; and { }. But this is usually not used, it is mainly intended to simplify machine generation of Haskell code (Hudak et al., 2007; O'Sullivan, Stewart and Goerzen, 2008, Chap. 9).
The Haskell98 report states following about the layout in Haskell:

> The meaning of a Haskell program may depend on its layout. The effect of layout on its meaning can be completely described by adding braces and semicolons in places determined by the layout. The meaning of this augmented program is now layout insensitive.
>
> The effect of layout is specified in this section by describing how to add braces and semicolons to a laid-out program. The specification takes the form of a function L that performs the translation. The input to L is:
>
> A stream of lexemes as specified by the lexical syntax in the Haskell report, with the following additional tokens: 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 zero 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 is preceded only by white space 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 $\{n\}$.

3.3 Introduction to Yi

Yi (Bernardy, 2008; Stewart and Chakravarty, 2005) is an editor, much like Emacs (Inc., n.d.) or Vim (Moolenaar, n.d.), but written in Haskell, with the goal of beging simple to configure and extend using Haskell. It comes with the option of using same key maps as Emacs or Vim.

Some of the provided features for Haskell editing in Yi is parenthesis matching, layout-aware edition, a GHCi interface and a cabal interface. Yi provides syntax highlighting using information from the lexer, highlighting keywords and Type/Data constructors in Haskell code. It also comes with support for highlighting several mainstream languages, such as C++, C, Perl and Python. A Javascript parser and lexer (Dogan, 2009) will be added in the next release.

Yi can statically or dynamically configure itself via a configuration file, which is compiled. In the configuration file, almost anything can be configured, similarly to the configuration file in xmonad (Stewart and Sjanssen, 2007).

Currently two frontends is supported, one using vty that only requires a terminal application and one using pango[2] that is under heavy development.

3.3.1 Lexing

Yi uses the Alex lexer generator(Marlow, n.d.) to produce a lexer that produces tokens. The produced tokens are used to feed the parser, which will produce an Abstract syntax tree (AST) this is visualized in fig. 12. The picture abstracts from the incremental behaviour details. The AST is then used in the end to highlight the text in a "highlighter".

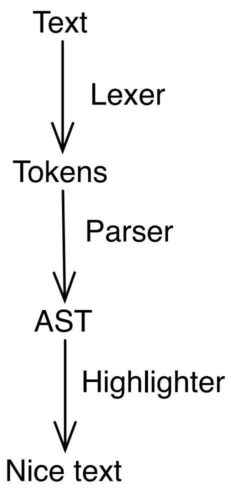


Figure 12: Lexing and parsing

Yi has an extra step when handling Haskell code (fig. 13) that inserts tokens marking the layout specifics such as where a block begins and ends.

3.3.2 Modes

The different parsers and lexers available in Yi are used by different so called modes that can be switched on and off via a simple key binding, similarly as in Emacs. Yi automatically sets mode depending on the file extension of the file. Files ending with hs will start with a haskell mode as default.

3.3.3 Highlighting

Apart from the different parse/lexer modes, the user also can choose from different sets of highlighting themes. It is easy to add a theme, all that needs

[2]From gtk2hs http://www.haskell.org/gtk2hs/

to be done is implementing the functions that take a token and colors it. A theme can be implemented completely in just a couple lines of code. The UIs support RGB colors and a color can be combined with different styles like italic and bold, coloring of the background and foreground of the text. The available number of colors and different styles depend on what frontend is being used.

In Yi the result from the parser is used when highlighting the code, Yi uses a stroke function and each structure that is used must be matched and highlighted in some way. In the stroke function errors marked by the parser are detected and highlighted as such. Yi provides different stroke colors that is dependant on which highlighting mode the user has chosen.

3.4 Parser Combinators

In this thesis an existing incremental parser library is used (Bernardy, 2009). The parser parses incrementally and caches earlier results so it will not have to reparse from the beginning at each keystroke. The parser library provides a solution to gracefully recovery from errors in the input. The parser library, which is inspired by the "Polish parser, step by step" article (John and Swierstra, 2003), implements similar combinators as described in that article.

```
• data Parser s a where
    Pure  :: a → a → Parser s a
    Appl  :: Parser s (b → a) → Parser s b → Parser s a
    Bind  :: Parser s a → (a → Parser s b) → Parser s b
    Look  :: Parser s a → (s → Parser s a) → Parser s a
    Shif  :: Parser s a → Parser s a
    Empt  :: Parser s a
    Disj  :: Parser s a → Parser s a → Parser s a
    Yuck  :: Parser s a → Parser s a
    Enter :: String → Parser s a → Parser s a
```

One can build parsers using the above constructors, but it is often more convenient to use standard Haskell mechanisms such as Alternative and Applicative. The *Parser* type implements these classes in the following way:

```
instance Applicative (Parser s) where
  (<*>) = Appl
  pure x = Pure x x
```

The (<*>) function performs sequential application and *pure* lifts a value to a Functor. The Functor instance of the *Parser* data type is simply

- **instance** *Functor* (*Parser s*) **where**
  fmap f = (*pure f* <∗>)

That gives us the *fmap* function, mapping a function on the given Functor. *fmap* has a synonym operator <$> that will be used in this thesis.

- **instance** *Alternative* (*Parser s*) **where**
  (<|>) = *Disj*
  empty = *Empt*

Disjunction lets us give a choice between possible parses, the disjunction is a *general choice* (Claessen, 2004) disjunction. The operator will force the parser to consider several possible paths if both possible paths can be valid more than zero tokens. As later will be described there are ways to affect when a path is discarded, if another one should be preferred.

- *many*, *some* :: *Parser s a* → *Parser s* [*a*]
  many v   = *some v* <|> *pure* [ ]
  some v   = (:) <$> *v* <∗> *many v*

many applies the same parser zero or more times. *some* applies the same parser one or more times.

- *symbol* :: (*s* → *Bool*) → *Parser s s*

symbol takes a function returning True if the token given should be shifted in the parser.

- *testNext* :: (*Maybe s* → *Bool*) → *Parser s* ()

testNext peeks at the next token of the input and produces a parser if the given function returns True for the token, otherwise another parse is chosen if possible. If no other parses are available the only option is to fail.

- $recoverWith :: Parser\ s\ a \to Parser\ s\ a$
  $recoverWith = Enter$ `"recoverWith"` $\circ$ $Yuck$

This rule should be used to recover from unexpected input, when a parser might fail. If the parsing rule is in a disjunction with the *recoverWith* parser, the *recoverWith* parser will be chosen to prevent a crash. Also the rule will insert a trace for debugging purposes. If only paths with *Yuck* is possible, then the one with least numbers of *Yuck* will be chosen. If the number of *Yuck* are the same after a threshold then one of the possible paths is chosen at random. This enables possibilities to write a parser that prefers something above something else, as long as the most preferable has least number of *Yuck*. The *Enter* data constructor is used to insert a trace in the parser, in this function the string `"recoverWith"` will be inserted.

4 The Haskell Parser

4.1 Overview

As earlier described, the text is first lexed into tokens, after this Yi inserts layout tokens into the list of tokens (fig. 13). The tokens that now contain information about the layout of the program can be parsed in the incremental parser, of course also the lexing and layout is done incrementally. The resulting AST can after the parsing be edited as will be shown in later sections. Finally the AST is used in a "highlighter" that adds color and style preferences such as if the text should be bold or italic.

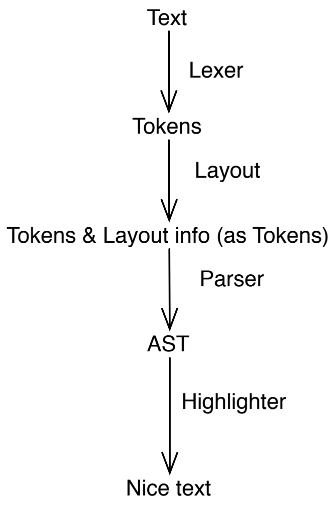


Figure 13: Lexing and parsing with layout

4.2 The lexer

As mentioned earlier Yi uses the Alex lexer generator, and has a lexer (Aho, Lam, Sethi and Ullman, 2006) for Haskell. The lexer divides the Haskell source code into different tokens listed in fig. 14.

```
data Token = Number | CharTok | StringTok | VarIdent | ConsIdent
    | Reserved ! ReservedType | ReservedOp ! OpType | Special Char
    | ConsOperator String | Operator String
    | Comment ! CommentType
    | THQuote
    | CppDirective | Unrecognized
```

Figure 14: Token data type

The tokens being used in the parser are

- *VarIdent* represents variable identifiers (*varid*)
- *ConsIdent* can be data types, data constructors etc. (*conid*)
- *Reserved* token contains all reserved words, like keywords. In Yi all keywords belongs to this group, including those that only are quasi-keywords. (*reserved* < *as*, *qualified*, *hiding*>).
- *ReservedOp* all of the predefined operators (*reservedop*)
- *Comment* this token contains *Open*, *Close* and *Line* to be able to match closing comments, opening and closing comments are separated. The matching of opening comment and closing is done in the lexer. This is important in the parser because of the need to allow comments everywhere.
- *CppDirective* Are as comments allowed everywhere and must be handled by the parser.
- *Special* contains layout based information that is inserted after the lexing phase.
- *ConsOperator* (*consop*) and *VarOp* (*varop*) together with *ReservedOp* (*reservedop*) they contain all operators.

Of course all tokens are used, but most of them are not relevant to the parser and are matched uniformly.

The lexer used by Yi had to be extended during this project in order to make it more precise. Since the previous parser in Yi only had parenthesis matching, there was no need to separate keywords etc. from each other. But since this project aims at parsing more precisely it is important to be able to identify different keywords (for example **module** and **import**), which is not possible without making the lexer more precise.

The following changes were made to the lexer in order to make it precise enough for the goal of this thesis

instance, **class**, **type**, **data**, **newtype**, **deriving**, **let**, **in**, **where**, **import**, *qualified*, *as*, *forall* and **module** got their own data constructors, the constructors were named *Instance*, *Class*... Instead of as previous being classified as *Reserved Other* they now belongs to *Reserved Instance*, *Reserved Class* etc.

In Haskell an ending # is allowed after *ConsIdent* and *VarIdent* when using the *magicHash* extension, this was previously not implemented in the lexer, but was implemented during this project in order to parse it correctly. Previously the following code

```
module A # where
```

was lexed into

```
(Reserved Other) ConsIdent (Operator "#") (Reserved Other)
```

but now it is lexed into

```
(Reserved Module) ConsIdent (Reserved Where)
```

4.3 Layout

Since Haskell is indentation sensitive, Yi has a Layout handler that helps separating the code into blocks based on the indentation. The layout is done after lexing the source code. A block is identified by a couple of added special tokens, which do not occur normally in Haskell. *Special* `'<'` (representing {), that begins a block and *Special* `'>'` (representing }), that ends a block.

The declarations are separated by the token *Special* `'.'` (representing ;). A block follows the keywords **where**, **let**, **do** and **of**. As an example after the lexing and layout phase the following code

```
module A where
times a = a * a
add a b = a + b
```

will look like this in tokens

```
(Special '<')
varIdent ConsIdent (Reserved Where)
(Special '<')
VarIdent VarIdent (ReservedOp Equal) VarIdent Operator VarIdent
(Special '.')
VarIdent VarIdent VarIdent (ReservedOp Equal)
  VarIdent Operator VarIdent
(Special '>')
(Special '>')
```

The code has more than one declaration (in this case two functions), so they are separated by the *Special* `'.'` token. A thing to note is that comments in Haskell do not need to be indented, so the layout does not care about comments. This will affect the parsing as later will be shown in section 4.4. As an example one can look at how the layout of these different code snippets are interpreted.

```
module A where
  -- comment maybe times is not needed
times a = a * a
```

the layout will do the following

```
(Special '<')
varIdent ConsIdent (Reserved Where)
(Comment Line)
(Special '<')
VarIdent VarIdent ...
```

In other words a **where** clause can be followed by some comments and then maybe a block. Another thing with the layout is that empty blocks is never created so consider removing the times function, then the layout would give following result.

```
(Special '<')
varIdent ConsIdent (Reserved Where)
(Comment Line)
(Special '>')
```

After a correct module declaration one can expect, either a block, a comment maybe followed by a block or nothing. During this project a couple of bugs has been detected in the layout handler and these have been fixed. An example of what has been changed, is that comments were taken into consideration in the layout when this project started.

The layout handler does not consider the already explicit layout that a user can provide, like GHC does[3]. A consequence of this is that the indentation information can be duplicated when explicit structuring is used.

4.4 The Haskell parser

The parser uses some different data structures to wrap the parse result in, they have been divided into 3 different parts. One for the module (*PModuleDecl*) declaration, one for imports (*PImport*) and one that contains everything else (*Exp*).

4.4.1 Basic Parsing Combinators

Some basic parser combinators has been implemented and will be used in later are:

- $sym \;\; :: (Token \rightarrow Bool) \rightarrow Parser\ TT\ TT$
  $sym\ f = symbol\ (f \circ tokT)$

  sym use the previously introduced *symbol* function to parse tokens of type *TT* (*Tok Token*).

- $exact :: [Token] \rightarrow Parser\ TT\ TT$
  $exact = sym \circ (flip\ elem)$

  exact parses, if possible, one of the tokens in the given list.

- $pEmpty :: Applicative\ f \Rightarrow f\ [a]$
  $pEmpty = pure\ [\,]$

  pEmpty lifts an empty list to be part of the *Applicative* class.

[3]GHC only inserts layout information when the user has not provided any

-
  ```
  please :: Parser TT (Exp TT) → Parser TT (Exp TT)
  please = (<|>) (PError <$> (recoverWith $ pure $ newT '!')
                         <*> pure (newT '!')
                         <*> pEmpty)
  ```

The *please* combinator can be used to prevent the parser from failing, if no other option is available it will use *recoverWith* to recover from the failure. If several parses containing *Yuck* are available in a disjunction, then the one with the least number of *Yuck* will be chosen. This function should always be used as an option if some token is to be expected, as an example, if the expected token is *Special* `')'` then *please* could be used in the following way *please* $ *pAtom* $ *exact* [*Special* `')'`]. If something else than the *Special* `')'` token appears, the error will be marked by inserting the token *Special* `'!'`.

During this project the following name convention has been used, rules parsing something begins with p, rules parsing something using *recoverWith* begins with pp (as in please parse).

```
data Exp t = ...
           PError { errorToken :: t
                  , marker     :: t
                  , comments   :: [t]
                  }
           ...
```

PError takes 2 tokens and a list of comment tokens, the first is the token to parse, the second is always the *Special* `'!'` token to indicate that an error has occurred. At last comes a list of comments that might follow the error.

-
  ```
  pErr :: Parser TT (Exp TT)
  pErr = PError <$> recoverWith (sym $ not ∘ uncurry (∨) ∘ (&&&)
         isComment
         (≡ CppDirective))
          <*> pure (newT '!')
          <*> pComments
  ```

pErr can be used to parse anything and marks the parsed token with the *PError* constructor. This rule is mostly used (sometimes in other rules) to collect all errors on a line (using *many*). The first token in *PError* should never be a comment, comments are never considered being an error.

- $pAtom, ppAtom :: [Token] \to Parser\ TT\ (Exp\ TT)$
  $pAtom = flip\ pCAtom\ pComments$
  $pCAtom :: [Token] \to Parser\ TT\ [TT] \to Parser\ TT\ (Exp\ TT)$
  $pCAtom\ r\ c = PAtom <\$> exact\ r <*> c$

pAtom is a rule to parse one of the tokens in the given list, followed by many comments (*pComments*). The reason for the structure of *pAtom* will be further explained in section 4.4.2. *ppAtom* is almost the same as saying $please \circ pAtom$ but it is implemented in the following way:

$ppAtom\ at = pAtom\ at <|> recoverAtom$
$recoverAtom :: Parser\ TT\ (Exp\ TT)$
$recoverAtom = PAtom <\$> (recoverWith\ \$\ pure\ \$\ newT$ `'!'`$)$
$\qquad\qquad <*> pEmpty$

This is done since the *PAtom* data constructor is expected. If *ppAtom* would be defined with *pAtom* i.e. $please \circ pAtom$ it would end up with the *Error* data constructor if the next token was another one than the expected. This would affect the pattern matching used later when setting the syntax highlighting. If the $please \circ pAtom$ would be used, two patterns can occur, either the *PAtom* or the *PError* constructor.

- $pTestTok :: [Token] \to Parser\ TT\ ()$
  $pTestTok\ f = testNext\ (uncurry\ (\lor) \circ (\&\&\&)\ isNothing$
  $\qquad (flip\ elem\ f \circ tokT \circ fromJust))$

pTestTok tests if the next token to parse is in the given list, if it is then the parse will succeed without shifting any tokens. This function will return a parser even if there are no more tokens to be parsed, in all other cases it will fail. The *isNothing* is used so that the parser does not fail if the result of the parser is *Nothing*.

- $pOpt :: Parser\ TT\ (Exp\ TT) \to Parser\ TT\ (Exp\ TT)$
  $pOpt\ r = Opt <\$> optional\ r$

When something optional is to be parsed, it is wrapped in the *Opt* constructor. The *optional* function, from the Applicative class, is used. The function simply parses the optional thing, or nothing and has the type $optional :: Alternative\ f \Rightarrow f\ a \rightarrow f\ (Maybe\ a)$

- $pToList :: Applicative\ f \Rightarrow f\ a \rightarrow f\ [a]$
  $pToList\ arg = (:)$ <\$> arg <∗> $pEmpty$

pToList put instances of the *Applicative* class into a list, the element provided will be the only element in the list.

-
  ```
  pSepBy :: Parser TT (Exp TT) → Parser TT (Exp TT)
             → Parser TT [Exp TT]
  pSepBy p sep = pEmpty
                   <|> (:) <$> p <*> (pSepBy1 p sep <|> pEmpty)
                   <|> pToList sep   -- optional ending separator
    where pSepBy1 r p′ = (:) <$> p′ <*> (pEmpty <|> pSepBy1 p′ r)
  ```

The *pSepBy* rule parses non-terminals (*p*) separated by some other (*sep*). Usually parsers use an *sepBy* rule that requires the pattern *p sep p* but *pSepBy* allows the pattern to end with a *sep* token (*p sep*).

This rule can as an example be used when parsing the export list in a module declaration or any function that is a list separated by a token with an optional ending separator. Exports are separated by a comma and as stated in the Haskell98 report the export list can have an optional ending comma.

-
  ```
  pParenSep :: Parser TT (Exp TT) → Parser TT (Exp TT)
  pParenSep = pParen ∘ (flip pSepBy pComma)
  ```

Since there are many occurrences where a list is separated by comma and placed inside of parenthesises we write a function for this pattern.

-
  ```
  pCParen, pCBrace, pCBrack
     :: Parser TT [Exp TT] → Parser TT [TT] → Parser TT (Exp TT)
  pCParen p c = Paren <$> pCAtom [Special '('] c
                  <*> p <*> (recoverAtom <|> pCAtom [Special ')'] c)
  ```

pCParen is used to match parenthesises, they are wrapped in the *Paren* constructor, comments can be allowed after the parentheses in this parse rule. The ending parenthesis is recognized by using *recoverAtom*.

The rules used to match braces and brackets are named *pCBrack* and *pCBrace*, they are constructed in the same way as *pCParen*. Brackets and braces share the same constructor as parentheses since they have the same behaviour and should be matched in the same way. To simplify there also exist *pParen*, *pBrack* and *pBrace* rules that parse with following comments.

$pParen, pBrace, pBrack :: Parser\ TT\ [Exp\ TT] \rightarrow Parser\ TT\ (Exp\ TT)$
$pParen = flip\ pCParen\ pComments$

- $startBlock, endBlock, nextLine :: TT$
  $startBlock = Special$ `'<'`
  $endBlock = Special$ `'>'`
  $nextLine = Special$ `'.'`

These rules are just a short hand for the layout matching.

- $pBlockOf' :: Parser\ TT\ a \rightarrow Parser\ TT\ a$
  $pBlockOf'\ p = exact\ [startBlock] *> p <* exact\ [endBlock]$

The above will parse p inside of a layout block.

- $pBlockOf :: Parser\ TT\ [(Exp\ TT)] \rightarrow Parser\ TT\ (Exp\ TT)$
  $pBlockOf\ p = Block <\$> (pBlockOf'\ \$\ pBlocks'\ p)$
  $pBlocks' :: Parser\ TT\ r \rightarrow Parser\ TT\ (BL.BList\ r)$
  $pBlocks'\ p = p$ \`$BL.sepBy1$\` $exact\ [nextLine]$

pBlockOf parses several p separated by *nextLine* tokens, inside of a layout block.

- **data** *Expr t* =

```
                ...
              | TC (Expr t)
              | DC (Expr t)
                ...
```

Is only used to wrap an expression, to mark that it should be colored with type constructor style or data constructor style.

4.4.2 Layout and error-recovery strategy

Haskell syntax depends on the indentation as discussed in section 3.2

```
id a = a
times b c = b * c
```

If the layout information is not taken into account it would be impossible to see where the first function ends and where the second begins since *times* can be another argument of *id*. A similar layout problem occurs when something is parsed and an error in the code is discovered, one would want the parser to show only the errors and not mark everything after the error as an error. Consider the following code.

```
module A b where
import D as C
```

The module has an error, namely the *b* character. When the parser comes to this point it expects either a parenthesized export list describing what the module exports, or the **where** keyword. But instead it gets a *b* character, now the parser must decide where the erroneous code ends. The reader with some Haskell experience will see that the following import is correct and should not be highlighted as an error, but the parser does not know how many tokens to shift. In other words the Parser needs something that indicates when to stop parsing erroneous code, one way would be to only parse one erroneous token, but there might be several of them. The solution to this problem is to assume that the layout information that has been inserted into the set of tokens always is correct.

A sound approach is to let the parser parse anything after an error as an error until the ”end of the line“ is reached. This is where the layout comes into the picture. Since information from the layout contains information about where a line ends, this can be used to make the parser shift tokens until it gets to the *nextLine* token. In this way the above code will highlight the *b* and **where** as errors and the follwing imports will be parsed correctly.

4.4.3 Handling comments

Comments in Haskell can appear anywhere in the code, and the parser must be able to cope with this. It is not possible to just discard the comments like compilers do, since the user expects the comments to be highlighted as comments in the code. Another problem when ignoring comments are that when an error is found, if the comments are not taken into consideration they would end up highlighted as an error, or not highlighted at all, but a comment should never be considered being an error. Both comment blocks and comment lines should be allowed anywhere, because of this the *isComment* rule was implemented, in order to be able to match any kind of comment.

One approach to this problem would be to simply split the list of tokens from the layout handler into tokens that are not comments and tokens that are comments and then in the end interleave the list. The problem with this approach is to interleave the list in the end without forcing a parse of the entire file. Consider the case when there is no comments. In order to interleave the two lists the head of the comment list must be checked, but if the parsed file has only one comment located in the end of the file, then all of the file must be parsed to see the head of the comment list.

Instead the solution used is to at first match the given token and after this match many possibly following comments, this is a list and can be empty it is done in the *pAtom* rule.

The solution used in this thesis to parse comments has one drawback, it is more costly to parse something followed by many comments than to parse something that can not have following comments. This will increase the number of possible parses and will affect the performance when Yi is used. In section 4.4.15 we present a solution that can be used to partially remove this drawback.

4.4.4 Parsing a module

The parsing can be divided into several parts that is parsed by their own parse rules. A file may begin with comments, module declaration, imports, functions, data declarations, type etc. what is allowed to follow depends on what comes before.

The data types in the parser is divided into 3 different categories, module, imports (imports are only allowed above the top declarations) and a body, containing the top declarations. The *PModule* data type is described in fig. 15.

It describes the different forms a Haskell module can take

- The module contains some comments maybe followed by some source code.
- The module contains comments, module declaration and possibly a body.
- The module contains a body, maybe with imports and some more content.

```
data PModule t
  = PModule { comments :: [t]
            , progMod  :: (Maybe (PModule t))
            }
  | ProgMod { modDecl :: (PModuleDecl t)
            , body    :: (PModule t)
            }
  | Body { imports      :: [PImport t]
         , content      :: (Block t)
         , extraContent :: (Block t)
         }
  deriving Show
```

Figure 15: Module data type

The data constructor *Body* contains two blocks after the list of imports, this is because of the layout done before parsing. If the layout after the module declaration is wrong, some code will be left in an own block outside of the module and it will not be highlighted. An example of such a layout is

```
module BL where
  times = (*)
add = (+)
```

which is transformed to the following list of tokens after layout is processed:

```
(Special '<')
(Reserved Module) ConsIdent (Reserved Where)
  (Special '<') VarIdent (ReservedOp Equal) ... (Special '>')
  (Special '.') VarIdent (ReservedOp Equal) ...
(Special '>')
```

but with the correct indentation all of the body is in the where block

```
...
(Special '<')
VarIdent . .   -- function times
VarIdent . .   -- function add
(Special '>')
(Special '>')   -- end of module block
```

The example with bad layout can not be compiled, but the parser should still parse everything outside of the block and if wrong then show this by marking it as an error. This layout "problem" occurs also in **where** blocks after functions, but as will be shown later this is solved in another way.

The content of a module, like data declarations, functions etc. all belongs to the *Exp* type, this since they can appear anywhere below the module declaration and imports without any order. Another way to do it would be to separate top declarations and expressions into different types.

```
pModule :: Parser TT (PModule TT)
pModule = PModule <$> pComments <*> optional
              (pBlockOf' (ProgMod <$> pModuleDecl
                                  <*> pModBody <|> pBody))
```

Figure 16: Rule for parsing a module

The *pModule* rule (fig. 16) parses many comments, followed by an optional block of code containing maybe a module declaration (*pModuleDecl*) followed by a body (*pModBody* if there is a module declaration or *pBody* if there is no module declaration).

4.4.5 Module declaration

To parse a module declaration, the first token must be *Reserved Module*, which represents the keyword **module**. After the module keyword a *ConsIdent* (the module name) is parsed.

The data type representing the module declaration has the form of fig. 17

```
data PModuleDecl t
   = PModuleDecl {keyword      :: (PAtom t)
                 , modName      :: (PAtom t)
                 , exports      :: (Exp t)
                 , whereKeyword :: (Exp t)
                 }
  deriving Show
```

Figure 17: Module declaration data type

PAtom t is the following shorthand

```
type PAtom t = Exp t
```

```
pModuleDecl :: Parser TT (PModuleDecl TT)
pModuleDecl = PModuleDecl <$> pAtom [Reserved Module]
                          <*> ppAtom [ConsIdent]
                          <*> pOpt (pParenSep pExport)
                          <*> ((optional $ exact [nextLine]) *>
                                (Bin <$> ppAtom [Reserved Where])
                                 <*> pMany pErr) <* pTestTok elems
  where elems = [nextLine, startBlock, endBlock]
```

Figure 18: Rule to parse a module declaration

When another token than the *ConsIdent* occurs after the **module** keyword, an error token will be inserted (due to the usage of *ppAtom*). After this follows optional exports, which is then followed by a **where** keyword that opens the body of the module. Before the end, many errors is allowed.

Three tokens can end a module declaration

nextLine if the *nextLine* token is parsed the declaration must be malformed, since the only way a *nextLine* token can appear at the end is if there is no where clause.

startBlock if the token is *startBlock* the declaration have a where clause that is non empty, or some other keyword that opens a layout block.

endBlock if the token is *endBlock*, the module declaration and the tokens above it is the only content of the module.

```
pExport :: Parser TT (Exp TT)
pExport = (optional $ exact [nextLine]) *> please
            (pVarId
             <|> pEModule
             <|> Bin <$> pQvarsym <*> (DC <$> pOpt expSpec)
             <|> Bin <$> (TC <$> pQtycon)
                      <*> (DC <$> pOpt expSpec))
  where pDotOp = (ReservedOp $ OtherOp "..")
        expSpec = pParen ((pToList $ please (pAtom [pDotOp]))
                                   <|> pSepBy pQvarid pComma)
```

Figure 19: Rule to parse exports

The exports are inside of parentheses and separated by commas, each export can be one of the following:

- *VarId* a function name
- An exported module
- A type constructor operator (extension to Haskell98) with optional exported data constructors
- A type constructor with optional exported data constructors

Before the open parenthesis an optional *nextLine* is parsed, this is due to the layout information. In Haskell98 there is no need to indent the exports so the following layout is fine

```
module Add
(add) where
```

The tokens given to the parser will have the following structure

```
(Special '<') (Reserved Module) ConsIdent
(Special '.') (Special '(')..
```

The same thing occurs when the where clause is located at the beginning of a new line, this is covered by having ((*optional* $ *exact* [*nextLine*])∗> before the expected **where**. This solution has a drawback, now the errors can stretch over three lines instead of the single one could expect. But the solution is better than showing that something is wrong when it is not, which would be the alternative. Of course it would be possible to decide that not indenting inside of the module declaration, is bad style and should be considered being an error even if it compiles fine.

After the *optional nextLine* the *please* function is used, due to the following possible scenario

```
module Add (
where
```

Since there is an opening parenthesis the parser will try to parse an export the *pSepBy* function allows empty exports, so this would have been an ordinary error if the layout of the **where** clause were different. The example will have the following token sequence after the Add *ConsIdent* (*Special* `'('`) (*Special* `'.'`) (*Reserved Where*) and because of the (*Special* `'.'`) an export is expected and if none is found, then the parser will fail without any possibility to recover. But by using please the parser inserts the *Special* `'!'` and recovers instead.

The *recoverWith* function when used here does not shift a token, instead it inserts a token to mark the error, because of this the error at the end is needed, consuming all input until it receives an end of the declaration token *nextLine*. This will have the effect that if someone writes an error after the **module** keyword, like following

```
module var (list) where
```

Everything following the **module** keyword will be parsed as an error, since the *var* does not match anything after the expected *ConsIdent*. On the other hand there is an advantage, if the programmer forget the *ConsIdent* like following code snippet

```
module (list) where
```

The parser will correctly interpret the exports and the **where** keyword, ending with a parse containing the extra token marking something went wrong, now it will be possible to mark the error by only highlighting one token.

4.4.6 Imports

The number of imports in a file is zero or more, fig. 20 describes how to parse several imports. The import declarations are separated by the *nextLine* token. *endBlock* will only occur if there is nothing following the imports, for example if the only content in the module is a couple of imports the last one will be finished by an *endBlock* token. The parser fails to recognize the case when an import ends with a ; like fig. 21, which is syntactically correct. The imports are parsed in this way since they are not allowed everywhere, imports are only allowed before the top declarations. Imports should be marked as errors if they occur at other places of the module.

```
pImports :: Parser TT [PImport TT]
pImports = many (pImport
                  <* pTestTok pEol
                  <* (optional $ exact [nextLine]))
  where pEol = [nextLine, endBlock]
```

Figure 20: Rule to parse several imports

One import (fig. 22) is parsed by first matching the keyword **import**, after this comes an optional *qualified* quasi-keyword. The *ConsIdent* is required and

```
import Data.List; import Data.Sequence;
import Data.Data
```

Figure 21: Three imports separated by ;

is parsed using the *ppAt*, so that the *recoverWith* function marks if is is missing. It is optional to have a synonym for the import, but if the *as* quasi-keyword is parsed, the following token must be a *ConsIdent*.

The import specification (*pImpSpec*) might contain the *hiding* quasi-keyword before the actual specification and this is allowed by using optional. If any errors follows, they will be caught by one of the three optional *pMany pErr* rules.

```
pImport = PImport <$> pAtom [Reserved Import]
             <*> pOpt (pAtom [Reserved Qualified])
             <*> ppAtom [ConsIdent]
             <*> pOpt (pKW [Reserved As] ppCons)
             <*> (TC <$> pImpSpec)
  where pImpSpec = Bin <$> (pKW [Reserved Hiding] $
                                     please pImpS) <*> pMany pErr
                        <|> Bin <$> pImpS <*> pMany pErr
                        <|> pMany pErr
        pImpS    = DC <$> pParenSep pExp'
        pExp'    = Bin <$> (PAtom <$> sym
                                     (uncurry (∨) ∘ (&&&)
                                     (flip elem [VarIdent, ConsIdent])
                                     isOperator) <*> pComments
                                      <|> pQvarsym)
                        <*> pOpt pImpS
```

Figure 22: Rule to parse single import

Since *qualified*, *as* and *hiding* which is not reserved is parsed in an import, they can safely be highlighted as keywords here and ordinary identifiers at all other occurrences. This means the problem of highlighting quasi-keywords has been solved.

4.4.7 Top declarations (body)

The parsing of the body is divided into several subsections. Since only **type**, **class**, **instance**, **data**, type signatures and the lhs of functions is allowed without any indentation, they are the first to be matched on the line. In the Haskell98 report this is described as the top declarations of a Haskell module. If the first thing that is not indented on a line is something else than a top declaration, it will be parsed as an error and thus can be highlighted as such.

A module consist of many top declarations and they are separated by the *nextLine* token. In Haskell code the top declarations can be parsed as in fig. 23.

```
pTopDecl :: [Token] → Parser TT [(Exp TT)]
pTopDecl at = pFunDecl at
                   <|> pToList pType
                   <|> pToList pData
                   <|> pToList pClass
                   <|> pToList pInstance
                   <|> pEmpty
```

Figure 23: Rule to parse top declarations

Where *pFunDecl* parse both lhs of functions and type signatures. *pFunDecl* also parse everything not covered by the other rules.

4.4.8 data

```
data Expr = ...
            | PData  {datakeyword     :: (PAtom t)
                     , context        :: (Exp t)
                     , types          :: (Exp t)
                     , rhs            :: (Exp t)
                     }
            | PData' {eqOrWhere       :: (PAtom t)
                     , cons           :: (Exp t)
                     , derivingKeyword :: (Exp t)
                     }
            ...
```

Figure 24: Data constructor for data declarations

The structure (fig. 24) for the data declarations has been divided into the lhs and rhs. This division into lhs and rhs simplifies support of parsing data declarations without a rhs.

After the **data** (fig. 25) keyword comes the optional context that will be marked as a type constructor using the *TC* constructor as wrapper. The *pSimpleType* parses patterns beginning with a parenthesis, or a type constructor, followed by several type variables.

```
pData :: Parser TT (Exp TT)
pData = PData <$> pAtom [(Reserved Data)]
             <*> pOpt (TC <$> pContext)
             <*> (TC <$> pSimpleType)
             <*> (pOpt (Bin <$> pDataRHS <*> pMany pErr))
             <* pTestTok pEol
```

Figure 25: Rule to parse a data declaration

As earlier mentioned the rhs is optional, and can be followed by some erroneous input that should be handled as errors. *pEol* is defined to be the marker that indicates that the data declaration has been ended.

The rhs begins either with = or a **where** depending if it is an ordinary data declaration or a GADT. The rule used to parse a data declaration can be viewed in fig. 25 and the right hand side of the data declaration in fig. 26.

```
pDataRHS :: Parser TT (Exp TT)
pDataRHS = PData' <$> pAtom eqW
                 <*> (please pConstrs
                         <|> pBlockOf' (Block <$> many pGadt
                         `BL.sepBy1` exact [nextLine]))
                 <*> pOpt pDeriving
                 <|> pDeriving
  where eqW = [(ReservedOp Equal), (Reserved Where)]
```

Figure 26: Rule to parse a rhs data declaration

The rhs of a data declaration is one of

- a GADT which is parsed by *pGadt* (fig. 27) this begins with the **where** keyword.
- a | separated list of data constructors, parsed by *pConstrs*, that begins with the same operator.
- a simple deriving (fig. 28) beginning with **deriving** keyword.

The rule for parsing GADTs (fig. 27) begins by parsing a type constructor, since the **where** keyword was already matched in fig. 26. An :: operator is required before the optional context, followed by the GADT content. *pAtype* parse type constructors, anything inside of parenthesis and anything inside of brackets.

A deriving declaration (fig. 28) is simply the **deriving** keyword followed by *please* either a type constructor or a parenthesized list of type constructors.

```
pGadt :: Parser TT (Exp TT)
pGadt = Bin <$> (DC <$> pQtycon)
            <*> (ppOP [ReservedOp $ OtherOp "::"]
                   (Bin <$> pOpt pContext <*>
                    (pTypeRhs <|> (pOP [Operator "!"] pAtype) <|> pErr)))
            <|> pErr
```

Figure 27: Rule to parse a GADT

```
pDeriving :: Parser TT (Exp TT)
pDeriving = TC <$>
                (pKW (exact [Reserved Deriving])
                 (please $ pParen
                  (Bin <$> please pQtycon
                    <*> many (Bin <$> pComma <*> please pQtycon))
                  <|> pQtycon))
```

Figure 28: Rule to parse deriving

4.4.9 type

The structure used for the type declarations (fig. 29) is part of the *Expr* type that is used for all different parts of the module body.

```
data Expr = ..
   | PType {typeKeyword :: (PAtom t)
            , typeCons     :: (Exp t)
            , typeVars     :: (Exp t)
            , eqOp         :: (PAtom t)
            , bType        :: (Exp t)
            }
   | ...
```

Figure 29: Data constructor for types

The rule used to parse type declaration begins with matching on the **type** keyword.

```
pType :: Parser TT (Exp TT)
pType = PType <$> pAtom [Reserved Type]
            <*> (TC <$> ppCons) <*> pMany pQvarid
            <*> ppAtom [ReservedOp Equal]
            <*> (TC <$> please pTypeRhs) <* pTestTok pEol
   where pEol = [startBlock
                ,nextLine
                ,endBlock]
```

Figure 30: Rule to parse a type declaration

4.4.10 class

The class declaration (fig. 31) is simply divided into the keyword, an optional context, a type constructor and a variable. After this comes an optional where clause which in this project is allowed to contain any of the top declarations.

```
data Expr = ...
          | PClass {classKeyword :: (PAtom t)
                   ,context      :: (Exp t)
                   ,typeCons     :: (Exp t)
                   ,typeVar      :: (Exp t)
                   ,whereBlock   :: (Exp t)
                   }
          ...
```

Figure 31: data constructor for a class declaration

4.4.11 instance

An instance declaration (fig. 32) is very similar to the class declaration, but here the first keyword is **instance** and the where clause is required. Also all top declarations are allowed inside of the where block of an instance declaration, this to simplify future support of extensions.

4.4.12 Function declaration

The left hand side of functions is parsed (fig. 33) by matching on either a parenthesis, a *VarIdent* or a keyword that is not yet parsed elsewhere at this stage. An example is the *foreign* keyword [4], used when using functions written in C.

[4]This is a keyword from an extension to Haskell98

```
data Expr = ...
          | PInstance { instanceKeyword :: (PAtom t)
                      , context         :: (Exp t)
                      , typeCons        :: (Exp t)
                      , typeVar         :: (Exp t)
                      , whereBlock      :: (Exp t)
                      }
          ...
```

Figure 32: Data constructor for an instance declaration

It is impossible to separate a function lhs from a type signature lhs on the first token, if the token is for example a *VarIdent*. To simplify things they are parsed together until the rhs shows up. If the rhs begins with :: we know it is an type signature, if a | or a = is found then we know we are parsing a function.

After the first token of a function there might come more parentheses, *VarIdent*s or keywords that is not yet handled, so the parser keeps parsing until it sees either a |, a = or a :: then the Right hand side of the function or type signature has been reached.

```
pFunDecl :: [Token] → Parser TT [(Exp TT)]
pFunDecl at = (:) <$> beginLine
                  <*> (pTypeSig
                          <|> pTr at)
                  <|> ((:)    <$> pAtom [Special ',']
                              <*> (pFunDecl at <|> pEmpty)))
  where beginLine = pCParen (pTr at) pEmpty
                    <|> pCBrack (pTr at) pEmpty
                    <|> (PAtom <$>
                           sym (flip notElem $ isNoise)
                           <*> pEmpty)
                    <|> (PError <$> recoverWith
                           (sym $ flip elem $ isNoiseErr)
                           <*> pure (newT '!') <*> pEmpty)
```

Figure 33: Rule to parse lhs of a function

at is a list of tokens not to parse as atoms. *beginLine* match the earlier mentioned allowed first tokens of the lhs in a function or type signature. *pTr* which is used, will parse anything that is not part of a top declaration (like **type**, **data**, **newtype** etc.).

4.4.13 Function right hand side

```
pFunRHS :: [Token] → Parser TT (Exp TT)
pFunRHS at = Bin <$> (pGuard
                        <|> pEq err at) <*> pOpt pst
  where pst = Expr <$> ((:) <$>
                        (PWhere <$> pAtom [Reserved Where]
                         <*> please (pBlockOf $ pFunDecl at))
                         <*> pTr′ at)
```

Figure 34: Right hand side of a function

A function rhs begins with either = or | and then follows the rest of the function, if it is a guard then more guards can follow. The implementation in fig. 34 separates rhs with guards (fig. 35) and equal signs because a rhs with equal signs is only allowed to have one equal, while guards can have several (one for each guard).

pGuard makes use of *some*, since the pattern repeats for every guard there must be one expression and then an equal sign and then another expression. In the Haskell98 report guards contain an *exp*, which is simply a = followed by an expression and more guarded *exp*, in this project they have been separated in a try to simplify the layout handling later on. *pTr′* will parse anything except pipe, where and equals, to prevent the parser from having several optional parses.

A function can always have an optional ending where clause containing more function lhss.

```
pGuard :: Parser TT (Exp TT)
pGuard = PGuard <$>
         some (PGuard′ <$> (pCAtom [ReservedOp Pipe] pEmpty) <*>
                   -- comments are by default parsed after this
               (pTr′ at)
               <*> please (pCAtom [(ReservedOp Equal)
                                  ,(ReservedOp RightArrow)]
                           pEmpty)
                   -- comments are by default parsed after this
                   -- this must be rightarrow if used in case
               <*> pTr′ at)
```

Figure 35: Parse guard part of functions

Comments are automatically parsed here because of the earlier discussed new solution to comment handling. This will be further described in the description of expression parsing. The parser allows both equal and right arrow after the first expression, this is so that in the future the *pGuard* parser can be used in **case** expressions with guards. *at* is as in *pFunDecl*.

In order to parse a function rhs beginning with a = the *pEq* function in fig. 36 is used

```
pEq :: [Token] → [Token] → Parser TT (Exp TT)
pEq at = RHS <$> (PAtom <$> exact [ReservedOp Equal]
                          <*> pEmpty)
                 <*> (pTr′ at)
```

Figure 36: Parse equal part of a function

4.4.14 Type signatures

In this project a quite simple and not very precise parse rule is used to parse type signatures, its main purpose is only to help color the type constructors in a correct way. Since type constructors in the type signatures only exist on the rhs, the lhs is not interesting for anything in this thesis.

```
pTypeSig :: Parser TT [(Exp TT)]
pTypeSig = pToList (TS <$> exact [ReservedOp (OtherOp "::")]
                          <*> (pTr noiseErr) <* pTestTok pEol)
   where pEol = [startBlock, endBlock, nextLine]
```

Figure 37: RHS of type signatures

The rhs type signature begins with *ReservedOp* (*OtherOp* "::") and then anything is allowed due to the fact that we are only interested in highlighting type constructors with correct color and this can easily be done with this information.

4.4.15 Expressions

When parsing expressions, most tokens is parsed as *PAtom* since we do not care about them, but the parenthesises are important since there should be matching done on them. Also as previously stated the **let** expressions will be parsed to be able to give indentation support. In expressions comments is always parsed as the first token in *PAtom* and after this follows an empty list, this can be done since there are no specific order being parsed that the comments can destroy. By using this solution the number of possible parse paths that needs to be considered can be lowered by one after each *PAtom*. Tokens that is considered

```
pTree :: [Token] → [Token] → Parser TT (Exp TT)
pTree at
    = (pParen ((pTr (at \\ [Special ',']))) pEmpty)
        <|> (pBrack ((pTr' (at \\ notAtom))) pEmpty)
        <|> (pBrace ((pTr' (at \\ notAtom))) pEmpty)
        <|> pLet
        <|> (PError <$> recoverWith
                (sym $ flip elem $ (isNoiseErr))
                 <*> pure (newT '!') <*> pEmpty)
        <|> (PAtom <$> sym (flip notElem (isNoise at)) <*> pEmpty)
where notAtom = [(Special ',')
                 , (ReservedOp Pipe)
                 , (ReservedOp Equal)]
```

Figure 38: Parsing expressions

being an error in expressions are defined by *isNoiseErr*, examples of things not allowed as expressions are **import**, **data** and **class**.

```
pLet :: Parser TT (Exp TT)
pLet = PLet <$> pAtom [Reserved Let]
          <*> (pBlockOf'
                  (Block <$> pBlocks
                   (pTr' at))
                   <|> (pEmptyBL <* pTestTok pEol))
          <*> pOpt (pCAtom [Reserved In] pEmpty)
  where pEol = [endBlock]
```

Figure 39: Parsing let expressions

In order to parse a let expression the *pLet* parse rule is used, it matches the beginning *Let* token and then comes a block or nothing followed by the end of a block. The *endBlock* is used since if there is no block to be matched, this means that the file must be empty after the **let** keyword.

```
Special '<'
Reserved Let
Special '>'   -- end of the block encapsulating all of the content
```

The optional **in** keyword is here only allowed at the end of a block, the layout automatically close the block when the in is found so this is no problem.

Several of the parse rules that has been implemented in this project has been left out of this report. The rules can instead be read in the source code, which is open and instructions how to get it is available in the Yi wiki.

4.5 Highlighting

In this project a decision was made that if something parsed contains an error, both the error and the first keyword on the line is also marked as an error. The reason behind this decision was that when something is missing in for example a data declaration like the following that does not contain any constructors

```
data Maybe a = a
```

there would not be anything to highlight since the error is that something is missing. Also this is consistent with the parenthesis matching, which will color the opening parenthesis if it does not have any closing parenthesis.

One drawback with this approach is that it forces the parser to some times parse further than what is visible, since if the keyword is to be marked with the error then all of the content that belongs to that keyword must be parsed to find possible errors. If an error is found, the first token is specially highlighted in red to indicate that an error has been found.

4.6 Testing the parser

Implementing a parser can be a tedious task if there is no testing tools available. It is easy to make some error causing the parser to fail or considering too many possible parses. In order to be able to parse online, the parser must be fast.

The test-suite must be able to use the parser as it is intended, token by token and being evaluated when each token is added, it should then continue from the previously evaluated state. Also the test-suite should have an option to parse the given file pushing all tokens to the parser so that it greedily parses the file.

When an error is found it is important to have an option to only feed a couple of tokens to the parser to find at which token it fails. In the visual process it is important to see where the parser shifts and what token it shifts. The information that can be interesting to see about the parser can be the complete process, an evaluated process and when a shift takes place in the parser. Also the possibility to insert trace messages is helpful.

The process tree produced by the test-suite can be visualized by dot (AT&T, n.d.). By using flags the test-suite can be run with different settings, the different options are

- -hs / -js tells the test to run Haskell files or Javascript files the default is to run a test with Javascript files.

- -cmp compares the result with a parse done by the simple parser that only does parenthesis matching. This is intended to be combined with the -hs flag.
- -oneBy pushes the tokens one by one as intended, this can be used to get a notion of how ”fast“ the parser is.
- -Toks=number where number is a number > 0, stops after pushing that many tokens to the parser.
- -r recursively search for files in the given folder.
- -Tree=file name tells the test to create a dot file with the given file name, this can then be visualized by running graphviz.

The order of the flags does not matter and the test-suite must be given either a folder or a file to parse, if a folder is given it will parse all files in the given folder with correct extensions.

An example of how things can go wrong (make the number of paths increase too much) is when some comments are allowed after some comments like in following example

```
parse :: Parser TT (Exp TT)
parse = exact [beginBlock]
            <* pAtom (exact [ConsIdent]) <*> pAtom (symbol $ cons True)
            *> exact [endBlock]
```

And the following code is parsed

```
List
  -- Comment
```

The possible parses can be viewed in fig. 40, which is the result of running the test-suite and printing a tree view of the result. The parser ends up with many possible parses, in fig. 40, LFail means the parser will fail if that branch is chosen, LSusp is a ”good“ state where the parse is successful. As can be seen, there are several paths that can be chosen to end up in a ”good“ state.

The implemented parser has been tested by running the test-suite on the source code of Yi (138 files) this will be further discussed in section 5.4. The most useful part of the test-suite has been the possibillity to run the parser on a larger amount of files without having to interact with it.

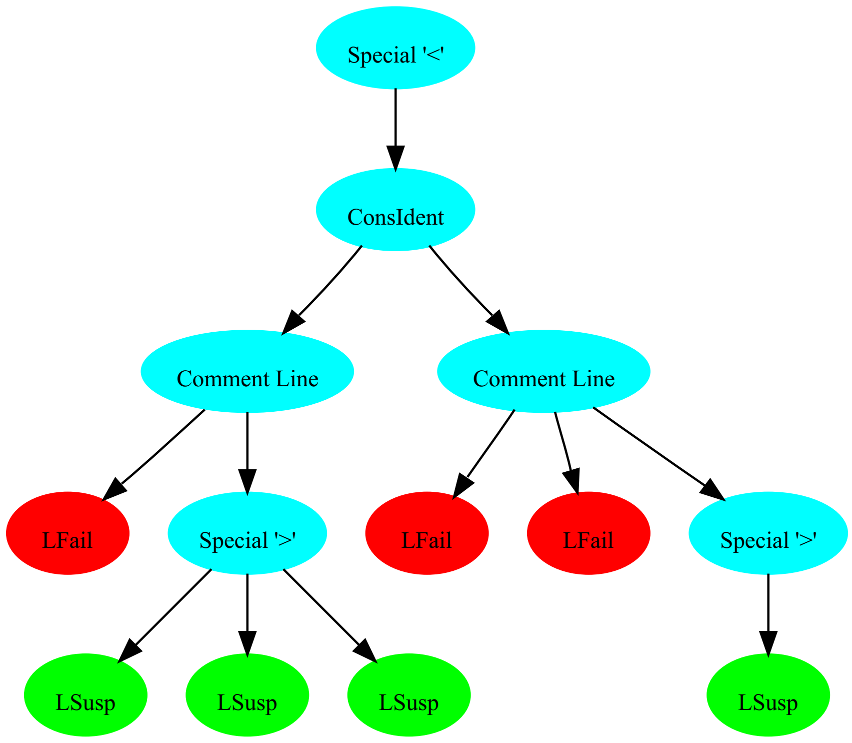


Figure 40: Bad Process

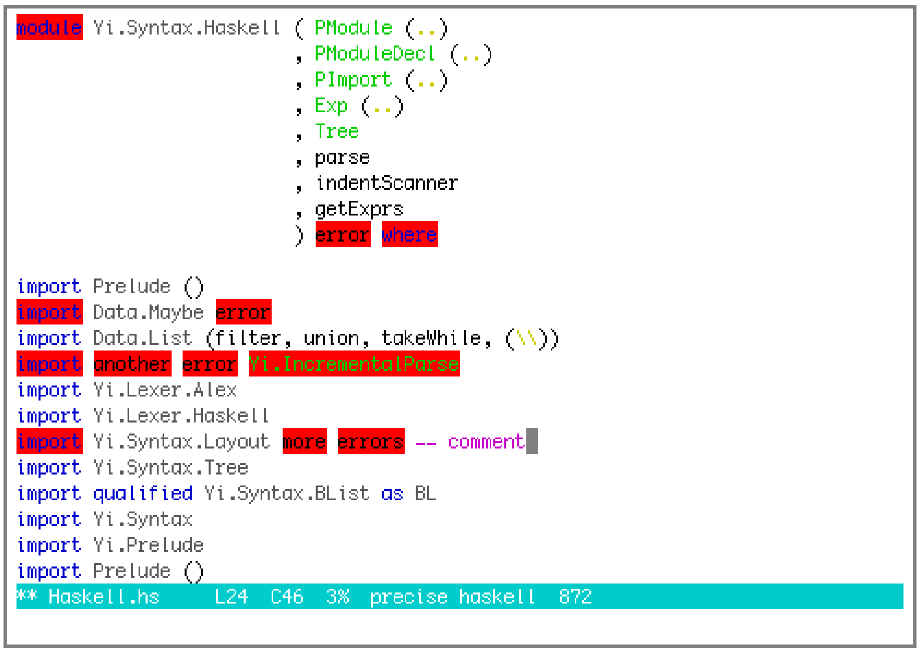
5 Result

The existing parser library has been improved during the project, in order to improve the parsing and to simplify the testing of implemented parsers.

5.1 Highlighting errors

The parser implements correct highlighting in all parsed constructs

- Correct highlighting of *as qualified hiding*
  if in imports they are highlighted as keywords and at other places they are highlighted as ordinary variables.
  see fig. 41
- Highlight errors when wrong in module declaration
  see fig. 41

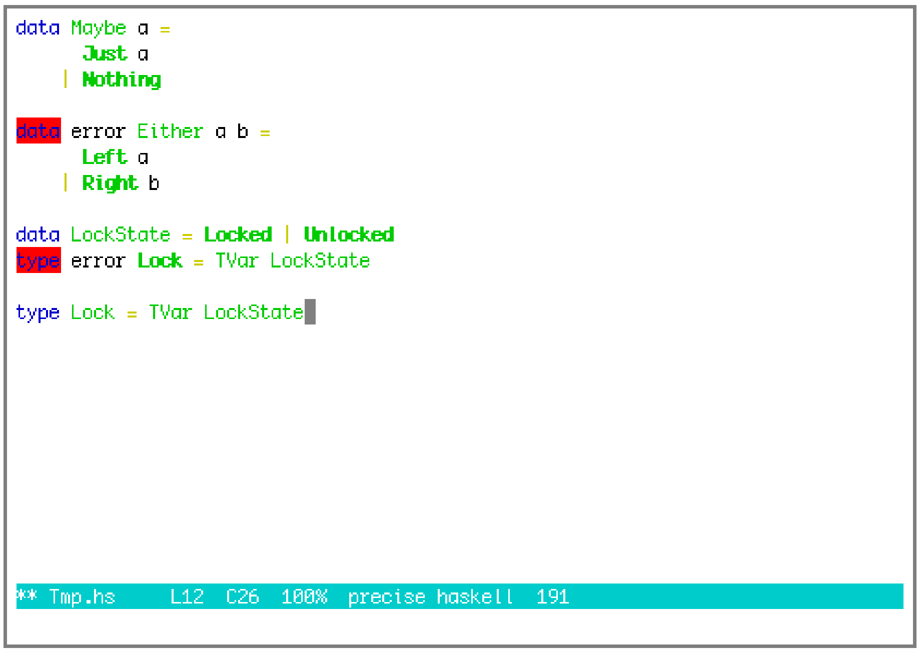


```
module Yi.Syntax.Haskell ( PModule (..)
                         , PModuleDecl (..)
                         , PImport (..)
                         , Exp (..)
                         , Tree
                         , parse
                         , indentScanner
                         , getExprs
                         ) error where

import Prelude ()
import Data.Maybe error
import Data.List (filter, union, takeWhile, (\\))
import another error Yi.IncrementalParse
import Yi.Lexer.Alex
import Yi.Lexer.Haskell
import Yi.Syntax.Layout more errors -- comment
import Yi.Syntax.Tree
import qualified Yi.Syntax.BList as BL
import Yi.Syntax
import Yi.Prelude
import Prelude ()
** Haskell.hs     L24  C46  3%  precise haskell  872
```

Figure 41: Highlighting of Module declaration and imports

- Highlight errors when wrong in import declarations
  see fig. 41
- Highlight errors when wrong in type declarations
  see fig. 42
- Highlight errors when wrong in data declarations
  see fig. 42
- Highlight errors when wrong in instance declarations
- Highlight errors when wrong in class declarations
- Type constructors is highlighted in their own color (separated from data constructors)
  Since type constructors is something else than data constructors they now have their own color to separate them
  see fig. 42
- Data constructors is highlighted in their own color (separated from type constructors)
  see fig. 42
- Imports is highlighted as modules
  The module name of the imported module is highlighted with the module name color
  see fig. 41



```
data Maybe a =
      Just a
    | Nothing

data error Either a b =
      Left a
    | Right b

data LockState = Locked | Unlocked
type error Lock = TVar LockState

type Lock = TVar LockState
```

```
** Tmp.hs      L12  C26  100%  precise haskell  191
```

Figure 42: Highlighting of data declarations, data constructors and type constructors

- Module name is highlighted in module name
  Same as with import module names
  see fig. 41

- Exported modules is highlighted in module name
  Same as above

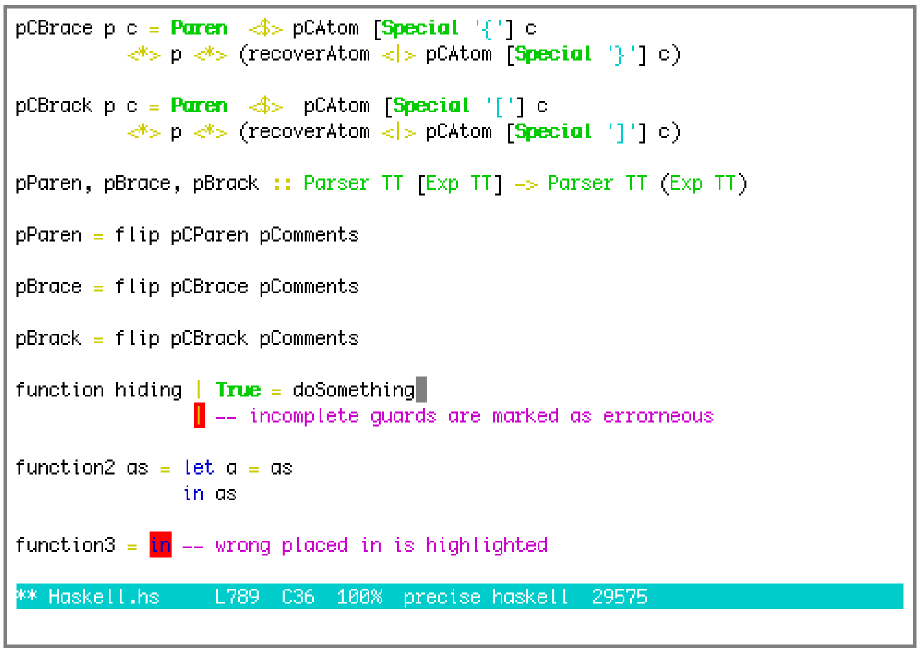
- **in** is highlighted correct in (most) occurrences
  in is highlighted as an error when not used correctly and tied to a **let** expression, since **in** is a real keyword it is only allowed in a **let** expression
  see fig. 43

- Incomplete guards in function rhs are highlighted as an error
  see fig. 43

5.2 Indentation

In addition to highlighting, the AST is available for any kind of functionality. In this section we briefly describe how it can be used to provide indentation hints to the user.

The indentation hints are very useful in a layout-aware language as Haskell. Instead of using space, it allows the user to "cycle" between different possible indentations. Indentation hints of guards, equal, where, of and parentheses are examples of what is currently supported. The *stopsOf* fig. 44 function runs



```
pCBrace p c = Paren  <$> pCAtom [Special '{'] c
          <*> p <*> (recoverAtom <|> pCAtom [Special '}'] c)

pCBrack p c = Paren  <$>  pCAtom [Special '['] c
          <*> p <*> (recoverAtom <|> pCAtom [Special ']'] c)

pParen, pBrace, pBrack :: Parser TT [Exp TT] -> Parser TT (Exp TT)

pParen = flip pCParen pComments

pBrace = flip pCBrace pComments

pBrack = flip pCBrack pComments

function hiding | True = doSomething
                | -- incomplete guards are marked as errorneous

function2 as = let a = as
               in as

function3 = in -- wrong placed in is highlighted

** Haskell.hs     L789  C36  100%  precise haskell  29575
```

Figure 43: Highlighting of guards, let expressions and hiding as variable

through the AST and based on the information, it provides a list of indentation hints. The indentation help is intended to help the programmer write easy to read code, and makes it easy to write code that has correct layout.

The parser allows the indentation to be so precise that it can provide different indentation when continuing a function on a new line. If it is an operator that is the continuation of the function on the new line as in fig. 45 then the <∗> operator is indented so that the end of it is where the equal sign ends. A function that has been split into two rows where the second row begins with something else than an operator is indented aligned with the end of the equal sign.

5.3 Language extensions

The extensions that has been implemented during this project is

GADT Fully supported.

EmptyDataDecls Fully supported.

RankNTypes The RankNTypes is only implemented for data declarations.

MagicHash This is only a lexer extension.

TypeOperators Partly implemented, type declarations and exports from a module, but not yet valid in data declarations.

```
stopsOf :: [Hask.Exp TT] → [Int]
stopsOf (g@(Hask.Paren (Hask.PAtom open _)
                ctnt (Hask.PAtom close _)) : ts)
    | isErrorTok (tokT close) ∨ getLastOffset g ⩾ solPnt
      = [groupIndent open ctnt]
        -- stop here: we want to be "inside" that group.
    | otherwise = stopsOf ts
        -- this group is closed before this line; just skip it.
stopsOf ((Hask.PAtom (Tok {tokT = t}) _) : _)
    | startsLayout t ∨ (t ≡ ReservedOp Equal)
    = [nextIndent, previousIndent + indentLevel]
      -- of; where; etc. ends the previous line.
      -- We want to start the block here.
      -- Also use the next line's indent:
      -- maybe we are putting a new 1st statement in the block here.
stopsOf ((Hask.PGuard' (PAtom pipe _) _ _ _) : _)
    = [tokCol pipe | lineStartsWith (ReservedOp Haskell.Pipe)]
      -- offer to align against another guard
stopsOf ((Hask.RHS (Hask.PAtom eq _) (exp : _)) : ts')
    = [(case firstTokOnLine of
      Just (Operator op) → opLength op (colOf' exp)
      _ → colOf' exp) | lineIsExpression] ++ stopsOf ts'
        -- offer to continue the RHS if this looks like an expression.
    ...
```

Figure 44: Function to get indentation suggestion

```
example a b cc = (,) <$> parseSomething
                 <*> parseSomethingElse
anotherExample = this function is quite long so
                   it has been split into two lines
```

Figure 45: Indentation of functions

5.4 Performance

Measuring performance in the parser by "typing" in the editor is not an easy task. Counting the number of paths available in the process gives a better measurement of the expected performance (if many possible paths are considered it will take more time).

The tests that have been run on the code source of Yi (138 Haskell files) indicates that the performance of the parser has almost as good performance as the parenthesis matching parser provided in Yi. A comparison between the parenthesis matching parser and the new improved parser can be viewed in fig. 46. The test shows that the number of possible paths that are considered in the precise parser is a bit less than twice as many as the number in the parenthesis parser. Considering that the precise parser parse more than twice as many things as the parenthesis parser this is quite a good result.

Some times has been measured when running the test-suite on all the files in the Yi project (those that are Haskell files). This was done to give the reader of this thesis an idea of how much time it takes to parse a file. Both pushing all content of each file to the parser and doing it token by token has been measured the result gives a hint if the parser is fast or not. The test-suite was run five times with token by token setting and five times with the push all tokens at once setting. The 138 files contain a total of 162150 tokens, which gives 1175 tokens in average per file.

#	**Old parser**	**New parser**	**Difference**
Possible paths	4802412	8124513	3322101
Paths per token	29.6	50.1	20.5
no flags	4.56s	8.5s	3.94s
oneBy flag	5.6s	11s	5.4s

Figure 46: Comparison between old parser, that only parsed parenthesises, and the new more precise parser

Pushing one by one has an approximated time of 11 seconds (measured by the terminal time command). This means that in average it spent 0.079s to parse all of the content in one file. When given all of the content at once it took approximately 8.5s which gives an average of 0.061s per file. When this test was performed, the test-suite was optimized by removing some of the output that is usually printed to the terminal.

During the last weeks of this project the developed mode has been the one used when developing the parser and the performance is good enough to be used interactively.

5.5 Future work

The future work that can be done on the parser is quite large. The parser should be extended so that most of the available language extensions are parsed. Maybe the parser should also adapt depending on which extensions are used. This would improve the parser speed since as few extensions as possible are considered.

The parser currently only highlights tokens that are wrong with a red color. It would be good if the parser provided some kind of error message indicating how the problem can be resolved.

Speed improvements are almost always possible, using the test program provided by this project it is possible to improve the speed by changing some parse rules. The discussed threshold where parses are discarded can be changed so that it can discard parses as early as possible.

A rule for **newtype** has not been implemented, but should be implemented in the future.

Other ways to use the AST, such as providing support to rename shadowing parameters, the usual solution to this problem is to add a ' to the parameters that is shadowed in the where clause.

```
f a b = g (a + b) (b - a)
  where g a c = a * c
  -- is translated into
f a b = g (a + b) (b - a)
  where g a' c = a * c
```

In the example a in the fixed version of f miss the ' by extending the parser so that the parameters of functions is parsed, it would be quite easy to implement a command that renames shadowed parameters in the where clause, using the information from the parser. Several extensions using the parser like this can be imagined, already a *dollarify* function exist replacing parenthesises with dollar when possible. Another one would be a command that makes the code point-free. This is a suggestions received from the Haskell-cafe mail list.

6 Conclusion

In this project a parser for Haskell98 has been developed, and it is possible to use it in an interactive editor. Along with this a test-suite has been developed. The parser can parse most of the Haskell98 syntax and mark where errors occur. Also some extensions to Haskell98 have been considered and can be parsed. The default for extensions that is not supported is that they are highlighted as errors.

Building a parser in a functional language has been a joy during this project, a functional language that is type safe as Haskell makes it easier to find the errors at compile time. Since most errors has been captured in the compiler, there has been almost no need for the test-suite checking for run-time errors that make the program exit. The only thing really needed to be tested has been if the parser considers too many different parses.

Goals that were stated in the beginning has been fulfilled:

Feedback online the parser library has been used and gives feedback quickly.

Precise feedback the parser find alot of the possible syntactical errors.

Robustness the parser recovers gracefully from errors.

Encapsulation done.

The technical goals:

Parse Haskell98 most parts is as described parsed.

Incremental parser is used.

Available in Yi done.

The implemented parser has been introduced to the Yi community and during the last weeks of the development the new mode using the parser was used while being developed.

References

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. (2006), Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley.

AT&T (n.d.), 'Graphviz', `http://www.graphviz.org/`.

Bernardy, J. (2008), Yi: an editor in haskell for haskell, in 'Proceedings of the first ACM SIGPLAN symposium on Haskell', ACM, Victoria, BC, Canada, pp. 61–62.

Bernardy, J. (2009), Lazy functional incremental parsing, in 'Proceedings of the second ACM SIGPLAN symposium on Haskell', ACM, Edinburgh, UK.

Claessen, K. (2004), 'Parallel parsing processes', Journal of Functional Programming **14**(6), 741–757.

Dogan, D. (2009), A JavaScript Mode for Yi, Master's thesis, Chalmers University of Technology.

Don, S. (n.d.), 'Yi', `http://www.haskell.org/haskellwiki/Yi`.

Hudak, P., Hughes, J., Jones, S. P. and Wadler, P. (2007), 'A history of haskell: Being lazy with class', ttt . http://dx.doi.org/10.1145/1238844.1238856.

Hughes, J. (1989), 'Why functional programming matters', Computer Journal **32**(2), 98–107.

Inc., F. S. F. (n.d.), 'GNU Emacs', `http://www.gnu.org/software/emacs/`.

John, R. J. M. and Swierstra, S. D. (2003), Polish parsers, step by step, in 'ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming', Vol. 38, ACM Press, pp. 239–248.

Marlow, A. (n.d.), 'Alex', `http://www.haskell.org/alex/`.

Moolenaar, B. (n.d.), 'Vim', `http://www.vim.org/`.

of Glasgow, U. (n.d.), 'Ghc', `http://www.haskell.org/ghc/`.

O'Sullivan, B., Stewart, D. and Goerzen, J. (2008), Real World Haskell, O'Reilly Media, Inc.

Peyton Jones, S. (2003), Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press.

Stewart, D. and Chakravarty, M. M. (2005), Dynamic applications from the ground up, in 'Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell', ACM Press, New York, NY, USA, pp. 27–38.

Stewart, D. and Sjanssen, S. (2007), Xmonad, in 'Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop', ACM, New York, NY, USA, p. 119.