Pire

Aiming to become Pi-forall's Refactorer, so far just another Pi-forall Parser

Andreas Reuleaux <`a.reuleaux@kent.ac.uk`>

November 2014

Table of Contents

Introduction 1
Pi-forall 101 1
Pire: Source code and Installation 3
Pi-forall's Parsec parser 4
The Trifecta parsers: a tour 8

Introduction

Pi-forall is a small dependently typed programming language developed by Stephanie Weirich, mainly for educational purposes, implemented in Haskell.

While less powerful than its better known cousins (Idris, Agda, Coq etc.), its internals may be more easily comprehensible than theirs, due to its small size (lines of code), but thanks also to Stephanie's great OPLSS lectures in 2013 [https://www.cs.uoregon.edu/research/summerschool/summer13/curriculum.html] and 2014 [https://www.cs.uoregon.edu/research/summerschool/summer14/curriculum.html], where she presented Pi-forall in a step by step manner. Thus there are several different versions available, and the one considered here is the final version, in the soln (=solutions) directory ie., of the code [https://github.com/sweirich/pi-forall/tree/2014] accompanying her OPLSS14 presentation.

Interesting therefore also as the target of some refactoring efforts, Pire aims to become Pi-foralls refactorer (pronounced any way you wish: "pi-re" for "Pi-forall refactorer", or just "pire" like "hire", and in French it sounds even worse :-)). At this point it is merely a different parser for Pi-forall, however:

- While Pi-forall's original parser is written in *Parsec*, Edward Kmett's *Trifecta* is used (as in Idris).
- Even though in most cases Trifecta could be used as a drop-in replacement for Parsec (the code could just be copied), there are some differences in the overall structure and handling of the parser: examples are given below, and layout parsing notably is handled differently (internally, stolen from Idris ie., rather than with LayoutToken.hs).
- More important, and that is really the whole point of this effort: Besides this reimplementation of Pi-forall's (abstract syntax) parser in Trifecta, there is a second parser that aims at parsing Pi-forall's concrete syntax, preserving white space and comments, and (together with a modified pretty-printer) should therefore be better suited for refactoring.

Pi-forall 101

Starting from just the bare minimum of language constructs:

```
a, A :=
  x             -- variables
  \x . a        -- lambda abstraction
  x b           -- function application
  (x: A) -> B   -- function type constructor (aka pi-types)
  x: Type       -- the type of types
```

Pi-forall is presented at various stages of increasing complexity in the course of Stephanie's discussion:

- More language constructs are added: type annotations, sigma types, propositional equality etc.
- The notion of (definitional) equality is refined from initially just alpha equivalence to beta-equivalence later.
- Constructors for new data types are introduced.

The language is a real dependently typed language:

- in that types are just expressions (by convention lower case letters are used for expressions, upper case letters for types)
- function type constructors (aka pi-types) are binders (the `x` in `(x: A) -> B` can appear in `B`)
- any type has type `Type` (the "type in type"-axiom), a simple rule to make Pi-forall's type system consistent (but inconsistent as a logic, not taking into account different universe levels).

The language is somehow limited:

- in that there are nor checks for totality or termination,
- there is no means to execute Pi-forall programs currently
- there are no (proof) tactics .

At the same time some aspects of Pi-forall are relatively elaborated:

- the notion of beta-equivalence for definitional equality eg., as mentioned above.
- Variables can be marked as *erasable*, ie. needed only for type checking, not at runtime. (Those are different from *implicit arguments*: there aren't any implicit arguments allowed in Pi-forall).
- A simple module system is provided.

Pi-forall's type system is certainly its most interesting part and in the center of Stephanie's discussion: how the typing rules can be broken down into corresponding checking and inference rules, and are reflected in the implementation.

One aspect of Pi-forall's *implementation* should be mentioned as well however: heavy use is made of the *Unbound* library internally, also developed by Stephanie Weirich, together with Brent Yorgey, which handles many things automatically that otherwise would be laboursome to implement:

- binding of variables (by means of Church encodings)
- alpha equivalence
- finding free variables
- substitution, and more.

Pire: Source code and Installation

While Pire could exist separately from Pi-forall (or at least rely on only parts of it), it was convenient to have all of Pi-forall's source code at hand.

The source code is therefore a complete copy of Pi-forall's source code from the solutions directory of Stephanie's OPLSS14 code, with Pire's modules added, all starting with `Re` (for "refactoring").[1]

Installation works for me in a cabal sandbox on a Debian jessie (= testing) system, ie. with ghc 7.6.3 currently, and relying heavily on quite some additional Haskell packages, installed not via cabal, but as Debian packages.

I find the Debian Haskell packages well maintained and easy to install, and enjoy the comfort of getting updates automatically once in a while, and am willing therefore to at times sacrifice having the newest version from hackage. This is reflected in the relatively relaxed version bounds of the packages required, eg. requiring trifecta to be just just >= 1.4.3, would make Debian's version still acceptable, even though there is a newer version: 1.5.1 already available on hackage. - I have relaxed the upper bounds for Pi-forall's original requirements as well: not sure if it was wise to do so, but will see.

Steps to install (roughly):

```
$ cd pire
$ cabal sandbox init
$ cabal update
$ cabal install highlight-versions
$ cabal --dry-run install | ./.cabal-sandbox/bin/highlight-versions
```

...and if happy with this dry-run, do the real installation:

```
$ cabal install
```

You should then be able to experiment with the original Pi-forall code, as well as Pire's in *cabal-repl* (the sandbox aware version of ghci), loading the main Pi-forall module e.g.:

```
$ cabal repl
...
> :l Main
```

Or Pire's trifecta parsers:

```
> :l ReParse
```

I plan to maintain the code in this "easy to install on Debian testing" manner, ie. not switch to Trifecta 1.5.1 eg. before version 1.5.1 enters Debian testing, unless there is a striking reason to use this newer version from hackage today already. The (zero length) files in the install directory I use merely as a reminder, which packages I installed from debian (`deb-*`), which via haskell cabal (`hs-*`, cf. the readme there for further explanations).

Pire has succussfully been installed on a Mac as well, and I would think that there are packages available on other systems similar to the Debian ones, and requirements, version numbers, bounds etc. should get some more tweaking in the future.

[1] Pire's cabal file `pire.cabal` started out as Pi-forall's cabal file as well, with some additional requirements (for Trifecta etc) added.

Pi-forall's Parsec parser

While Pi-forall's parser is arguably its least interesting part (and barely mentioned in Stephanie's lectures) parsing is the starting point for any serious implementation (and hence refactoring) work.

Moreoveer, as the absy parser developed here is merely a rewrite of Pi-forall's *Parsec* parser in *Trifecta*, some examples, of how this Parsec parser can be used, may be worthwhile looking at, both as a Parsec recap, and to get in first touch with some Pi-forall code.

Say we are given a small Pi-forall input file `Sample.pi` with some function declarations:

```
-- -*- haskell -*-
-- Sample.pi

module Fac where

import Nat

-- data Nat : Type where
--   Zero
--   Succ of (Nat)

two : Nat
two = Succ (Succ Zero)

-- adapted from nat_eq in Nat
-- cf also http://mazzo.li/posts/AgdaSort.html

nat_leq : Nat -> Nat -> Bool
nat_leq = \ x y .
  case x of
     Zero -> True
     Succ m -> case y of
            Zero -> False
            Succ n -> nat_leq m n

fac: Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))

foo: Nat -> Nat
foo = \n. fac (mult 2 n)

bar: Nat -> Nat
bar = \n. let fac = ((\x . plus x 2) : Nat -> Nat) in plus (foo n) (fac (mult 3 n))
```

Never mind too much the details of those declarations at this point, just note a few points:

- The factorial function `fac` is defined the usual way, and requires the notion of natural numbers (`Nat`) together with some operations on them (`mult`, `minus`). Those we import from the `Nat` module, not shown here, but available with Pi-forall's source as well.
- We need so few things from `Nat` here, that we could have defined them ourselves, the commented out `Nat` data type definition gives a taste of what they'd look like, and indeed: doing so was useful when developing the Trifecta parser, as parsing a single file obviously is a easier than handling module imports.
- Pi-forall allows for Haskell-style comments.
- The remaining functions may serve as examples for some simple refactorings later.

- `nat_leq` (less or equal on naturals) and `two` exercise the `Nat` data type, and note how the `case` expression uses layout syntax (as in Haskell), ie. requires layout parsing.

- The very first `-*- haskell -*-` line was just a means for me to get some (Haskell) syntax highlighting in Emacs for Pi-forall programs.

The usual way of interacting with Pi-forall is calling its `goFilename` function in `Main.hs`, which parses, typechecks, and pretty-prints the given Pi-forall source file, in the case of `Sample.pi` above:

```
*Main > goFilename "/home/rx/etc/th-p/pi/Sample.pi"
processing Sample.pi...
Parsing File "/home/rx/etc/th-p/pi/Nat.pi"
Parsing File "/home/rx/etc/th-p/pi/Sample.pi"
type checking...
Checking module "Nat"
Checking module "Fac"
module Fac where
import Nat
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n (1)))
foo : Nat -> Nat
foo = \n . fac (mult (2) n)
bar : Nat -> Nat
bar = \n .
        let fac = \x . plus x (2) in plus (foo n) (fac (mult (3) n))
*Main >
```

As can be seen:

- the output is quite readable and resembles the input, but all the comments are lost, as are the exact levels of indentation, newlines and parenthesis. The result is therefore of limited use for any refactoring efforts only, and this was the reason for developing another white space aware parser.

- Automatic conversions between different representations, like the one above for `Nat` from `Succ (Succ Zero)` to `2` in the function `two`, we don't want either for our refactoring purposes.

- The fact that our `Sample.pi` is also typechecked, cannot really be seen in the output above (except for the messages `type checking...` etc.), but is important for we don't want to concern ourselves with programs that are syntactically, but not type correct, and type checking gives us a means to ensure that.

While `goFilename` is a convenient way to parse, typecheck and pretty-print our Pi-forall code all in one go, sometimes we want more fine grained control, want to see intermediate results, the abstract syntax tree ie..

The following examples are meant as hints, how to "look under the hood" and discover more details, rather than as a complete description of the system.

Given that our `Sample.pi` file imports the `Nat` module, we cannot use the parsing functions from `Parser.hs` directly, but only those from `Module.hs` (namely `getModules`). Cheating at the source code of `goFilename` we can get the abstract syntax tree of our `Sample.pi` file by running `getModules` in the `Control.Monad.Error` monad, like so (only the beginning of the result is shown):

```
*Main > runErrorT (getModules ["/home/rx/etc/th-p/pi/"] "Sample.pi")
Parsing File "/home/rx/etc/th-p/pi/Nat.pi"
Parsing File "/home/rx/etc/th-p/pi/Sample.pi"
Right [Module {moduleName = "Nat", moduleImports = [], moduleEntries = [Data "Nat"
 Empty [ConstructorDef "/home/rx/etc/th-p/pi/Nat.pi" (line 14, column 3) "Zero"
 Empty,ConstructorDef "/home/rx/etc/th-p/pi/Nat.pi" (line 15, column 3) "Succ" (Cons
 Runtime _ (Pos "/home/rx/etc/th-p/pi/Nat.pi" ...
```

The result is a list of modules, really: `Nat.pi` and `Sample.pi`, in the Either monad. Normally we'd be interested only in the *last* module parsed (`Sample.pi` ie.), and `fromRight'` from the `either` package is handy for revealing this result (again, only the beginning of the output is shown):

```
*Main > :m +Data.Either.Combinators
*Main Data.Either.Combinators > runErrorT (getModules ["/home/rx/etc/th-p/pi/"]
 "Sample.pi") >>= return . last . fromRight'
Parsing File "/home/rx/etc/th-p/pi/Nat.pi"
Parsing File "/home/rx/etc/th-p/pi/Sample.pi"
Module {moduleName = "Fac", moduleImports = [ModuleImport "Nat"], moduleEntries = [Sig
 two (Pos "/home/rx/etc/th-p/pi/Sample.pi" (line 12, column 7) (TCon "Nat" [])),Def
 two ...
```

Now we can use *disp* for pretty printing the result:

```
*Main Data.Either.Combinators > runErrorT (getModules ["/home/rx/etc/th-p/pi/"]
 "Sample.pi") >>= return . disp . last . fromRight'
Parsing File "/home/rx/etc/th-p/pi/Nat.pi"
Parsing File "/home/rx/etc/th-p/pi/Sample.pi"
module Fac where
import Nat
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n .
        if nat_leq n 0 then 1 else mult n ((fac ((minus n (1)))))
foo : Nat -> Nat
foo = \n . fac ((mult (2) n))
bar : Nat -> Nat
bar = \n .
        let fac = ((\x . plus x (2)) : Nat -> Nat) in
        plus ((foo n)) ((fac ((mult (3) n))))
*Main Data.Either.Combinators >
```

If we were looking at the contents of just a single self-contained file, without any imports of other modules ie., say at a file `SimpleSample.pi`:

```
-- -*- haskell -*-
-- SimpleSample.pi

module Fac where


data Nat : Type where
  Zero
  Succ of (Nat)

two : Nat
two = Succ (Succ Zero)


nat_leq : Nat -> Nat -> Bool
nat_leq = \ x y .
  case x of
     Zero -> True
     Succ m -> case y of
            Zero -> False
            Succ n -> nat_leq m n

fac: Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))
```

then we could use `parseModuleFile` from `Parser.hs` directly. (We need to provide `emptyConstructorNames` from `Syntax.hs`, so that the parser can collect information about data types and constructors found (`Nat`, `Zero`, `Succ`,...) running in the `State` monad internally):

```
*Main Syntax Data.Either.Combinators > runErrorT (parseModuleFile  emptyConstructorNames
 "/home/rx/etc/th-p/pi/SimpleSample.pi") >>= return . disp . fromRight'
Parsing File "/home/rx/etc/th-p/pi/SimpleSample.pi"
module Fac where
data Nat : Type where
  Zero
  Succ of (_ : Nat)
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n .
        if nat_leq n 0 then 1 else mult n ((fac ((minus n (1)))))
*Main Syntax Data.Either.Combinators >
```

The situation is less complicated (does not require any file I/O) if we just want to parse a string as an expression:

```
*Main Data.Either.Combinators > parseExpr   "   \\x  .  a  "
Right (Pos "<interactive>" (line 1, column 4) (Lam (<(x,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 11) (Var a))))
*Main Data.Either.Combinators > fromRight' $ parseExpr   "   \\x  .  a  "
Pos "<interactive>" (line 1, column 4) (Lam (<(x,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 11) (Var a)))
*Main Data.Either.Combinators >
```

Again, pretty printing is convenient for reading the result:

```
*Main Data.Either.Combinators > disp $ fromRight' $ parseExpr   "   \\x  .  a  "
\x . a
*Main Data.Either.Combinators >
```

`Parser.hs` exports only `parseModuleFile` and `parseExpr`, but we can exercise the individual building blocks of the parser of course, by loading `Parser.hs` directly: ie. we can parse expresssions, declarations, signature definitions, variables etc. (shown is parsing an expression):

```
*Parser > runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v})
 [] "<interactive>" " Succ n") emptyConstructorNames)
Right (Pos "<interactive>" (line 1, column 2) (App (Var Succ) (Var n)))
*Parser >
```

This time we have to "unpeel the onion" of our monad transformer stack ourselves: the aforementioned *StateT* keeps track of constructor names, and *FreshM* from `unbound` is responsible for names and variable bindings.

Note how, without any further knowledge of *Succ*, *Succ n* is just parsed as a function application. This may not be what we want: An admittedly ad-hoc `prelude` helps recognizing *Succ* as a data constructor rather than as a regular function:

```
*Parser > let prelude = ConstructorNames (S.fromList ["Nat"]) (S.fromList ["Zero",
 "Succ"])
*Parser > runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v})
 [] "<interactive>" " Succ n") prelude)
Right (Pos "<interactive>" (line 1, column 2) (DCon "Succ" [Arg Runtime (Var n)] (Annot
 Nothing)))
*Parser >
```

The Trifecta parsers: a tour

Load the parser module `ReParse` to start some experiments:

```
reuleaux@softland ~/etc/pire $ cabal repl
Preprocessing library pire-0.1...
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
...
Loading package HUnit-1.2.5.2 ... linking ... done.
[1 of 9] Compiling LayoutToken      ( src/LayoutToken.hs, interpreted )
[2 of 9] Compiling Syntax           ( src/Syntax.hs, interpreted )
[3 of 9] Compiling Parser           ( src/Parser.hs, interpreted )
[4 of 9] Compiling PrettyPrint      ( src/PrettyPrint.hs, interpreted )
[5 of 9] Compiling Environment      ( src/Environment.hs, interpreted )
[6 of 9] Compiling Equal            ( src/Equal.hs, interpreted )
[7 of 9] Compiling TypeCheck        ( src/TypeCheck.hs, interpreted )
[8 of 9] Compiling Modules          ( src/Modules.hs, interpreted )
[9 of 9] Compiling Main             ( src/Main.hs, interpreted )
Ok, modules loaded: Syntax, Parser, TypeCheck, Main, LayoutToken, Environment,
 PrettyPrint, Equal, Modules.
*Syntax > :l ReParse
[1 of 5] Compiling Syntax           ( src/Syntax.hs, interpreted )
[2 of 5] Compiling PrettyPrint      ( src/PrettyPrint.hs, interpreted )
[3 of 5] Compiling ReSyntax         ( src/ReSyntax.hs, interpreted )
[4 of 5] Compiling RePP             ( src/RePP.hs, interpreted )
[5 of 5] Compiling ReParse          ( src/ReParse.hs, interpreted )
Ok, modules loaded: Syntax, PrettyPrint, ReParse, ReSyntax, RePP.
*ReParse >
```

Parse an identifier:

```
*ReParse > parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> identifier)
 piInit ) "  wow this rocks"
"wow"
```

Now parse the identifier in the white space aware parser, ie. including the whitespace following it

```
*ReParse > parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> id')
 piInit ) "  wow this rocks"
("wow",Ws " ")
*ReParse >
```

Naming conventions are as follows:

- Parsers for abstract syntax (absy parsers) have names like the original parsec ones: *identifier*, *variable*, *reserved*, *expr* etc. and they parse the original data structures defined in `Syntax.hs`. Their code is in most cases identical to the Parsec versions.

- Primed versions of those names: *identifier'*, *expr'* etc. are used for the corresponding concrete syntax parsers, but sometimes I have taken the liberty to shortcut / simplify those names: *id'* for *identifier'*, *res'* for *reserved'* etc., especially for those parsers that are used very often (the building blocks). They parse the white space aware data structures in `ReSyntax.hs`

- Any other variation of a name, like double primes: *id''* denote helper functions, experimental versions etc.

For reading the code it may be easiest to start with the simple parsers (below the comment line `-- the real stuff - parsing` in `ReParse`) and then proceed to the more complicated ones, and later maybe read the Trifecta setup etc before that line.

More complicated expressions can be parsed, too:

```
*ReParse > parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr)
 piInit ) "   \\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "
Pos "<interactive>" (line 0, column 4) (Lam (<(n,{Annot Nothing})> Pos
 "<interactive>" (line 0, column 9) (If (Pos "<interactive>" (line 0, column 12) (App
 (App (Var nat_leq) (Var 0@0)) (DCon "Zero" [] (Annot (Just (TCon "Nat" []))))))) (Pos
 "<interactive>" (line 0, column 29) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot
 (Just (TCon "Nat" []))))] (Annot (Just (TCon "Nat" []))))) (Pos "<interactive>" (line 0,
 column 36) (App (App (Var mult) (Var 0@0)) (Paren (Pos "<interactive>" (line 0, column
 44) (App (Var fac) (Paren (Pos "<interactive>" (line 0, column 49) (App (App (Var minus)
 (Var 0@0)) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot (Just (TCon "Nat" []))))]
 (Annot (Just (TCon "Nat" []))))))))))))) (Annot Nothing))))
parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr) piInit ) "   \
\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "minus n 1))  "
  :: MonadIO m => m ()
*ReParse >
```

And the equivalent expression in concrete syntax (*expr′* instead of *expr*):

```
*ReParse > parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr')
 piInit ) "   \\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "
Lam' (Ws "") (<(n,{Annot' Nothing (Ws " ")})> If' (Ws " ") (App' (App' (Var' nat_leq
 (Ws " ")) (Var' 0@0 (Ws " "))) (DCon' (DCName' "Zero" (Ws " ")) [] (Annot' (Just
 (TCon' (TCName' "Nat" (Ws "")) [])) (Ws ""))))) (Ws " ") (DCon' (DCName' "Succ" (Ws
 " ")) [Arg' Runtime (DCon' (DCName' "Zero" (Ws "")) [] (Annot' (Just (TCon' (TCName'
 "Nat" (Ws "")) [])) (Ws "")))] (Annot' (Just (TCon' (TCName' "Nat" (Ws "")) [])) (Ws
 ""))) (Ws " ") (App' (App' (Var' mult (Ws " ")) (Var' 0@0 (Ws " "))) (Paren' (ParenOpen
 (Ws "")) (App' (Var' fac (Ws " ")) (Paren' (ParenOpen (Ws "")) (App' (App' (Var'
 minus (Ws " ")) (Var' 0@0 (Ws " "))) (DCon' (DCName' "Succ" (Ws "")) [Arg' Runtime
 (DCon' (DCName' "Zero" (Ws "")) [] (Annot' (Just (TCon' (TCName' "Nat" (Ws "")) []))
 (Ws "")))] (Annot' (Just (TCon' (TCName' "Nat" (Ws "")) [])) (Ws "")))) (ParenClose (Ws
 "")))) (ParenClose (Ws "  ")))) (Annot' Nothing (Ws "")))
*ReParse >
```

Of course one wants some kind of pretty printing here, but *parseTest* is only so useful here, because it does its own job of rendering the result by running in the IO monad, as can be seen by its type its type:

```
  *ReParse > :t parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr)
 piInit ) "   \\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "
parseTest (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr) piInit ) "   \
\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "
  :: MonadIO m => m ()
  *ReParse >
```

Thus one wants to use one of the other Trifecta parsing functions: `parseString`, `parseFromFile`, `parseFromFileEx`, eg. parse the if-then-else expression above with `parseString` yields a result in a Result type [2]

[2] note, that we need here as well:

- *piInit*, a kind of empty prelude (to keep track of constructer names, help with layout parsing etc.)
- *top*, a means to tell the parser, where we start, at the top ie.

```
*ReParse > parseString (runInnerParser $ runFreshMT $ evalStateT (whiteSpace *> expr)
 piInit) top " \\n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))  "
Success (Pos "<interactive>" (line 1, column 2) (Lam (<(n,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 7) (If (Pos "<interactive>" (line 1, column 10) (App
 (App (Var nat_leq) (Var 0@0)) (DCon "Zero" [] (Annot (Just (TCon "Nat" []))))) (Pos
 "<interactive>" (line 1, column 27) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot
 (Just (TCon "Nat" []))))] (Annot (Just (TCon "Nat" []))))) (Pos "<interactive>" (line 1,
 column 34) (App (App (Var mult) (Var 0@0)) (Paren (Pos "<interactive>" (line 1, column
 42) (App (Var fac) (Paren (Pos "<interactive>" (line 1, column 47) (App (App (Var minus)
 (Var 0@0)) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot (Just (TCon "Nat" []))))]
 (Annot (Just (TCon "Nat" []))))))))))))) (Annot Nothing)))))
*ReParse >
```

This result can then be displayed:

```
*ReParse > disp $ fromSuccess $ parseString (runInnerParser $ runFreshMT $ evalStateT
 (whiteSpace *> expr) piInit) top " \\n . if nat_leq n 0 then 1 else mult n (fac (minus n
 1))  "
\n . if nat_leq n 0 then 1 else mult n ((fac ((minus n (1)))))
*ReParse >
```

So what?, you might say. - See what happens if we parse the same expression with some comments in between: First with the expr parser as above:

```
*ReParse > disp $ fromSuccess $ parseString (runInnerParser $ runFreshMT $ evalStateT
 (whiteSpace *> expr) piInit) top " \\n . if {-foo-} nat_leq n 0 then {-bar-} 1 else mult
 n (fac (minus n 1))  "
\n . if nat_leq n 0 then 1 else mult n ((fac ((minus n (1)))))
*ReParse >
```

The comments are lost - they are comments after all for the absy parser. But now parse them with white-space aware parser (`expr'` instead of `expr`):

```
*ReParse > disp $ fromSuccess $ parseString (runInnerParser $ runFreshMT $ evalStateT
 (whiteSpace *> expr') piInit) top " \\n . if {-foo-} nat_leq n 0 then {-bar-} 1 else
 mult n (fac (minus n 1))  "
\n . if {-foo-} nat_leq n 0 then {-bar-} 1 else mult n (fac (minus n 1))
*ReParse >
```

See how they are kept!

OK, lots more stuff to explain, and to do of course, just a rough overview at this point:

- explain some more advanced usage of the white-space aware parser, that is possible already (some examples can be found in `exampls.hs`, and be used by cut & paste at the cabal repl prompt).
- in particular explain how to parse files with `parseFromFile`, `parseFromFileEx`, thereby sticking to simple cases first (no imports of other modules), but import of other modules is in the works.
- create some test cases, to be run automatically.
- refine the parsers, improve the situation for module imports and make sure, they work with all the original Pi-forall .pi files.
- get rid of some of the desugaring that happens in the parser currently still (eg. conversion of Nats)
- implement some kind of forget function: from concrete syntax to abstract syntax, thereby allowing typechecking (which requires abstract syntax).

- explain how layout parsing is handled.
- explain, how we have leveraged unbound's substitutions for some simple refactorings (but have a look at other binding libs as well.)
- clean up the code (ie. remove some of my traces of my experimention)
- Have some fun, and do some refactorings!