Pire - Pi-forall's Refactorer

Andreas Reuleaux

Table of Contents

Introduction

Pi-forall is a small *dependently typed* programming language developed by Stephanie Weirich, mainly for educational purposes, and 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, 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 (not to mention numerous other great presentations of hers), cf. section #p4a_101 for some more details about *Pi-forall*.

There are thus several different versions of *Pi-forall* available, and the one considered here is the final version (in the *final* directory) of the code [https://github.com/sweirich/pi-forall/tree/2014] accompanying her OPLSS14 presentation (in *Pi-forall*'s *2014* branch, its main branch i.e.).

Interesting therefore also as a target for *refactoring*, *Pire* aims to be *Pi-forall*'s *refactorer* (pronounced any way you like: "pi-re" for "Pi-forall re-factorer", or "pire" like "hire", or as we do: in French, where it sounds even worse :-)).

Some noteworthy points:

- Whereas *Pi-forall*'s original parser is written in *Parsec*, *Pire* is built upon Edward Kmett's *Trifecta*.

  While often we could use *Trifecta* as a drop-in replacement for *Parsec*, our parser differs from *Pi-forall*'s original parser:

  - layout notably is handled differently (the offside rule ie.):
    - we use (Michael Adam's) *indentation-trifecta*,
    - rather than (*Parsec*'s) *LayoutToken.hs*, as used in *Pi-forall*.
  - We provide several different parser versions:
    - our regular parser parses *expressions* (parser *expr*), *declarations* (parser *decl*), *patterns* (parser *pattern*), *matches* (parser *match*), and so on,
    - we provide *white space aware* variants of these parsers as well (*expr_*, *decl_*, *pattern_*, *match_*, etc), parsing (*absy*, *token tree*) pairs, *token trees* being faithful representations of the original source code encountered (preserving white space and comments), ie. we parse *concrete syntax* rather than *abstract syntax*, and thus can preserve the programmer's original program layout (including comments).
    - in an older implementation we had kept white space within the absy:
      - our *expr* parser would parse absy expressions (*Lam*, etc.),
      - *expr_* would parse white space aware expressions then (*Lam_*, etc.)

      (*Lam* and *Lam_* both being *expression* constructors). - While focusing on the above (*absy*, *token tree*) pair approach these days, we keep these ideas in the *old syntax* modules *OldSyntax*, *OldParser*, *OldNavigation* etc.

- *Pi-forall* is available within *Pire* nevertheless, in the *PiForall* name space.

- *Pire* builds upon (Edward Kmett's) *Bound* library, rather than on (Stephanie Weirich's) *Unbound*, as *Pi-forall* does.

- navigation in the syntax tree is possible then:
  - with *lenses* for simple movements,
  - and by means of *zippers* in a more general fashion (our *zippers* being built upon *functions*, we can leverage *lenses*).

- We *calculate* line column information (rather than *parse* it), from the *concrete syntax*, that we keep anyhow.

- *Renaming* is one of the classical refactorings then, that our refactoring tool provides, more of them yet to come.
- Last, but not least: we advantage of *dependent types* for some more ambitious tasks, *List-to-Vector* refactorings notably.

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-type)
  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*, Σ*-types*, *propositional equality* etc.,
- the notion of *(definitional) equality* is refined from initially just α-equivalence to β-equivalence,
- constructors for new *data types* are introduced.

Pi-forall has all the characteristics of a *dependently typed* language:

- *types* depending on *values*,
- types are *first class*,
- internally types are just *expressions* (by convention lower case letters are used for *expressions*, upper case letters for *types*),
- the function type constructor (aka *Π*-type) is a *binder* (the `x` in `(x: A) → B` can appear in `B`),
- and so is the introduction form for Σ*-types* (dependent pairs), among others,
- any type has type *Type* (the *type in type*-axiom), a simple rule to make *Pi-forall*'s type system consistent (while inconsistent as a *logic* still: not taking different universe levels into account).

The language is somehow limited:

- there are no *checks* for *totality* or *termination* (*termination* and *totality* being the price for *propositions as types*, commonly know as the as the *Curry-Howard isomorphism*),
- *execution (evaluation)* does not feature prominently in *Pi-forall*: while a basic means to execute *Pi-forall* programs is provided with reduction to WHNF, there is no *REPL* for example,
- there are no (proof) tactics,
- there are no *implicit arguments* in *Pi-forall*, making *Pi-forall* code verbose at times.

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

- the notion of β-equivalence for definitional equality for example, as mentioned above,

- variables can be marked as *erasable* (needed for type checking only, not at runtime - and by the way: *erasable arguments* are different from *implicit arguments*),
- 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 two different groups: *checking* and *inference rules*, as reflected in the implementation.

Pire: Source code and Installation

Installation instructions as of December 2018 (ghc 8.4.3, cabal 2.2), some sections yet to be refined.

Pire consists of

- its main (library) section,
- a test suite (of several individual tests),
- documentation (its *haddocks* or API docs, and *orgmode/html/pdf* docs)

and includes *Pi-forall*'s original source code in particular. *Pire* is organised in several *sub-projects*[1] (sub-directories):

- *pi-forall* contains *Pi-forall*'s original source code (its *full* distribution),
- *pire-pi-forall* makes *Pi-forall* available in the *PiForall* name space then (cf. #p4a below for details and examples)
- *pire-docs* in directory *docs*

In theory it should be possible to install (and use) *Pire* with basic *git* and *cabal* commands only, cf. section #standard_installation below.

In practice it may be easier, however, to use our *shake* tool chain instead (which we provide as well for convenience, cf. section #shake), the *shake* commands being useful for *Pire development* in particular.

standard installation with *git* and *cabal*

clone *Pire*, and create a sandbox

```
git clone git://git.a-rx.info/pire
cd pire
cabal sandbox init
```

update the available packages, and do a dry run first, to see, what packages would be installed

```
cabal update
cabal install --dry-run
```

> tip: maybe use *highlight-versions* to see the details in a more fine grained manner (in colour)
>
> ```
> cabal update
> cabal install highlight-versions
> cabal install --dry-run | highlight-versions
> ```
>
> for example

[1] (*sub-projects* being a feature of *cabal 2.x*, recognised by the *cabal new-** commands, and organised in file *cabal.project*).

```
~/work/p2 $ cabal install --dry-run | highlight-versions
Resolving dependencies...
In order, the following would be installed (use -v for more details):
NoTrace-0.3.0.3
bound-2.0.1
getflag-1.0
indentation-core-0.0.0.2
raw-strings-qq-1.1
show-prettyprint-0.2.2
indentation-trifecta-0.1.0
pire-0.2.5
~/work/p2 $
```

and if you are happy with this dry run, do the real installation:

```
cabal install
```

continue exploring *Pire* (cf. section #simple_examples below), and exercise *cabal*'s possibilities, depending on interest for example:

- run *Pire*'s test suite, cf. section #testing below, ie. install with `--enable-tests` in particular, and provide flags as necessary for individual tests:

  ```
  cabal install --enable-tests -f "discover" -f "roundtrip" -f "doctest" ...
  ```

  run the tests then, one by one

  ```
  cabal test utests
  cabal test doctests
  cabal test roundtrip
  ```

  or all of them together

  ```
  cabal test
  ```

  etc.

- work with individual subprojects/sections, with *unit tests* for example: `cabal new-repl utests`
- create the *haddock* (API) documentation by installing with `cabal install --enable-documentation`, and running `cabal haddock` then
- load *Pi-forall* within *Pire*, cf. #p4a below, etc.

simple examples (new syntax)

If your installation went well, you should be able to use *Pire* on the *cabal repl*[2] command line.

```
cabal new-repl
```

Tip: depending on your installation (with or without *testing* enabled e.g.) it may be necessary to specify *cabal*'s target more precisely (*pire* is our main target):

[2] We use the general term *cabal repl* to mean its newer form *cabal new-repl* in particular.

```
cabal new-repl pire
```

Try parsing some *expressions* for example (or create them with *smart constructors*: *lam* etc.), pretty print them, etc., start by loading some *Pire* modules:

```
>>> :m +Parser NoPos
```

or load a predefined list of *Pire* modules (`_pire` is a laundry list of further imports, for convenience):

```
>>> :l _pire
```

then continue (get rid of position information with *nopos*):

```
>>> parse expr   "\\x  .  a  "
...

>>> nopos $ parse expr   "\\x  .  a  "
LamPAs [(RuntimeP,"x",Nothing)] (Scope (V (F (V "a"))))

>>> pp $ nopos $ parse expr   "\\x  .  a  "
\x . a

>>> lam "x" $ V "z"
Lam "x" (Scope (V (F (V "z"))))

>>> pp $ lam "x" $ V "z"
\x . z

>>> nopos $ parse expr  "if cond then    a else   b"
If (V "cond") (V "a") (V "b") Nothing

>>> pp $ nopos $ parse expr  "if cond then    a else   b"
if cond then a else b
```

You may want to try out *Pire*'s *white space aware parsers* (use *expr_* instead of *expr* for *expressions*, and likewise for *declarations*, *matches* etc), which all parse (*absy*, *token tree*) pairs:

```
>>> nopos $ parse expr_  "\\x  .  a  "
(LamPAs [(RuntimeP,"x",Nothing)] (Scope (V (F (V "a")))),Node [Pair (Token "\\") (Ws
 ""),Node [Pair (Binder "x") (Ws "  ")],Pair (Token ".") (Ws "  "),Abstract ["x"] (Scope
 (Pair (Id (F (Id "a"))) (Ws "  ")))])

>>> nopos $ parse expr_  "if cond then    a else   b"
(If (V "cond") (V "a") (V "b") Nothing,Node [Pair (Token "if") (Ws " "),Pair (Id "cond")
 (Ws " "),Pair (Token "then") (Ws "    "),Pair (Id "a") (Ws " "),Pair (Token "else") (Ws
 "   "),Pair (Id "b") (Ws "")])

>>> ppr $ fst $ nopos $ parse expr_  "if cond then    a else   b"
If (V "cond") (V "a") (V "b") Nothing

>>> ppr $ snd $ nopos $ parse expr_  "if cond then    a else   b"
Node [Pair (Token "if") (Ws " ")
     ,Pair (Id "cond") (Ws " ")
     ,Pair (Token "then") (Ws "    ")
     ,Pair (Id "a") (Ws " ")
     ,Pair (Token "else") (Ws "   ")
     ,Pair (Id "b") (Ws "")]

>>> pp $ fst $ nopos $ parse expr_  "if cond then    a else   b"
if cond then a else b
```

```
>>> pp $ snd $ nopos $ parse expr_  "if cond then    a else   b"
if cond then    a else   b
```

etc. Building upon these parsers, *navigation* in the syntax tree, and *renaming* are the next steps in your journey of exploring refactorings.

old syntax

Maybe you want to try out *Pire*'s *old syntax* (defined in modules *OldSyntax*, *OldParser*, *OldSmart*, *OldNavigation*, etc.), where white space is kept within the *absy*.

The *old syntax* can be explored with the `-DOLDSYNTAX` flag (with `:l _pire` loading the same predefined list of *Pire* modules as above, but with *OldSyntax* instead of *Syntax*, *OldParser* / *Parser* etc.).

```
>>> :set -DOLDSYNTAX
>>> :l _pire
...

>>> ppr $ nopos $ parse expr_  "if cond then    a else   b"
If_ (IfTok "if" (Ws " "))
    (Ws_ (V "cond") (Ws " "))
    (ThenTok "then" (Ws "    "))
    (Ws_ (V "a") (Ws " "))
    (ElseTok "else" (Ws "   "))
    (Ws_ (V "b") (Ws ""))
    (Annot Nothing)

>>> pp $ nopos $ parse expr_  "if cond then    a else   b"
if cond  then    a  else    b

>>> nopos $ parse expr_  "\\x  .  a  "
LamPAs_ (LamTok "\\" (Ws "")) [(RuntimeP,Binder_ "x" (Ws "  "),Annot Nothing)] (Dot
 "." (Ws "  ")) (Scope (Ws_ (V (F (V "a"))) (Ws "  ")))

>>> pp $ nopos $ parse expr_  "\\x  .  a  "
\x  .  a

>>> ppr $ nopos $ parse expr_  "\\x  .  a  "
LamPAs_ (LamTok "\" (Ws ""))
        [(RuntimeP,Binder_ "x" (Ws "  "),Annot Nothing)]
        (Dot "." (Ws "  "))
        (Scope (Ws_ (V (F (V "a"))) (Ws "  ")))
```

shortcuts

Tip: These are handy shortcuts thus for *cabal repl*

- to either load the (default:) *new syntax* modules,
- or the *old syntax* modules

(put them in your `~/.ghci`):

```
:{
let cmdP _ = return $ unlines
             $ ""
             :":l _pire"
             :[]
     :}
```

```
-- aka "pire"
:def p cmdP


:{
let cmdOldP _ = return $ unlines
             $ ""
             :":set -DOLDSYNTAX"
             :":l _pire"
             :[]
      :}

-- aka "old pire"
:def oldp cmdOldP
```

load *Pire* modules then:

- with

  ```
  $ cabal new-repl
  ```

  ```
  >>> :p
  ```

  for *Pire*'s (default:) *new syntax*

- or with

  ```
  $ cabal new-repl
  ```

  ```
  >>> :oldp
  ```

  for *Pire*'s *old syntax* flavor.

We haven't found a convenient way yet to *unset* the `-DOLDSYNTAX` flag, i.e. having loaded *Pire*'s *old syntax*:

```
>>> :oldp
```

switch back to *Pire*'s *new syntax* :

- `:unset -DOLDSYNTAX` does not work
- and `:set -DOLDSYNTAX=False` causes a lot of warnings,

our advice for now is thus to quit the *cabal repl* and start anew:

```
$ cabal new-repl
```

```
>>> :p
```

shake tool chain

The *shake* tool chain builds upon *cabal*, and allows (among other things):

- to issue the standard *cabal* commands,
- build our *cabal* files (recursively, if need be), and possibly with additional dependencies, cf. below,
- build the documentation (the *haddocks* or API docs, as well as the *orgmode/html/pdf* docs)

- clean up our code base,
- work recursively on all our subprojects,
- do testing: run individual tests, or all of them together, possibly with the `--oldsyntax` flag (for the *old syntax* modules), etc.

and there are various good reasons to use it, for development, or just for building *Pire*:

- The *shake* commands and flags are simpler / shorter usually than their *cabal* counterparts, for example:
  - (*shake*) `--tests` for (*cabal*) `--enable-tests`
  - (*shake*) `--docs` for (*cabal*) `--enable-documentation`
  - the *shake* target *bootstrap* will issue several *cabal* (sandbox) commands:

    ```
    cabal sandbox init
    cabal update
    ```

  - the cabal option `--haddock-options=--no-print-missing-docs` is conveniently set, when building the *haddocks* with *shake* target *hd*
  - etc.
- (this point needs updating:) *Pire* being split in several *sub-projectes* (directories, cf. file `cabal.project`), this directory structure is recognised by the newer *cabal* commands only (cabal *new-build*, *new-repl*, etc.), but as of this writing (*cabal 2.0* in debian testing), not all of the standard *cabal* commands have *new* equivalents: *cabal new-install* for example is not available in cabal 2.0 (but only in *cabal* >= 2.2).

old cabal commands	*new cabal* 2.x commands
install	*new-install* in cabal >= 2.2
repl	*new-repl*
etc.	

- It was convenient to experiment with slightly different *cabal* files at times: we try to keep (sub-project) dependencies to a minimum, but it could be useful for example to require *turtle* during development as well (needed for some administrative task of ours, but certainly not a dependency of *Pire* core), additional dependencies can be requested with flags `--tools`, `--moredeps`, `--morecode`, `--turtle`, `--foldl`, among others.
- We may want to install the dependencies for *testing*, but *not* have *cabal* run our test suite at installation time, etc.

We therefore offer our *shake* tool chain as well as an option for building *Pire* (files *Build.hs*, *Boot.hs*, *build.sh* and their dependencies: *Tools.hs*, *TShake.hs*, *Flags.hs* etc.), and discuss the *shake* commands, that we typically use for installation, and other administrative tasks (cf. section #standard_installation above for the more standard *cabal* commands).

We only require that the *cabal* files distributed with *Pire* via *git* are built without any extra flags:

- ie. use any additional dependencies of your liking during experimentation,
- but make sure, that `pire.cabal` is built with

```
./build.sh -B -V cb
```

  - possibly with `-b` / `-boot` (using `Boot.hs`),
  - but without `--turtle`, `--tools`, `--moredeps`, for example

- likewise for the other cabal files `docs/pire-docs.cabal`, `pire-pi-forall/pire-pi-forall.cabal`, build them all recursively with

  ```
  ./build.sh -R -B -V cb
  ```

  (and maybe `-b` / `-boot`)

- *Pire* should be buildable (testable), of course, with the standard *cabal* commands, cf. section #standard_installation above (ie. the dependencies in `pire.cabal` should be sufficient for installing and testing *Pire*, but not for editing our *shake* files maybe, and not for building our *orgmode* docs).

Here is a list of tasks (and flags), that we perform with *shake* typically, as well as some background information (organising principles):

- ```
  ./build.sh -boot ...
  ```

  (or `-b` for short)
  - uses the `Boot.hs` *shake* file
  - for basic installation, and administrative tasks,
  - written in pure *shake* (ie. no dependencies on *turtle*)

- ```
  ./build.sh ...
  ```

  (without `-boot` / `-b`)
  - applies the `Build.hs` *shake* file,
  - for more advanced tasks (but basic `Boot.hs` tasks are reexported as well)
  - requires *turtle*, and can be used thus after installation (of *turtle*) only.

- Our debian system provides *shake* packaged as *libghc-shake-dev*, but *turtle* we have to install from *hackage*, thus we can bootstrap/install from `Boot.hs`, but not from `Build.hs`, nevertheless we like to use *turtle*, whenever we can (and prefer *turtle* over *shake* at times).

  We prefer to use *turtle* unqualified in *Build.hs* (for ease of experimentation with *turtle*), and therefore have to take measures to avoid name clashes (*shake* and *turtle* both offer *Shell* for example). `Build.hs` is unusual for a *shake* file then, in that we import only some of *shake*'s functions unqualified (*TShake.hs* is bridging *turtle* and *shake*), in practice our `Build.hs` file works fine.

- typical usage is thus to start with the bootstrapping/installation commands from *Boot.hs* (with `-boot` / `-b`), and then use regular shake commands from *Build.hs* (without `-b`), possibly recursively (with `-R`) :

  ```
  ./build.sh -boot -R -B -V ...
  ./build.sh       -R -B -V ...
  ```

- `Flags.hs` are shared by `Boot.hs` and `Build.hs` and several more modules (even though not every flag makes sense for every command).

- `-B` is for *shake*'s `--rebuild`, `-V` for *verbose*,

- `-rec` / `-R` applies *shake* recursively
- target *cb* for building *cabal* files, eg.

```
./build.sh -b -R -B -V cb
```

- `--turtle`, `--foldl`, `--interpolate`, `--missingh`, among others, for additional individual dependencies,
- `--tools` for tool dependencies, `--moredeps` for some more (often used) dependencies,
- `--morecode` to include source code in directory *morecode* as well
- running individual tests:
  - `--roudtrip` for setting our cabal file *roundtrip* flag, and thus enable *roundtrip* testing,
  - likewise `--doctests` for our *doctests*,
  - `--discover` for our *discover* / *testsuite*,

  cf. the `pire.cabal` file for more details/flags.

Here is a sample session of ours:

clean out our code base (to start anew)

```
./build.sh -boot -R -B -V cl
```

bootstrap a sandbox

```
./build.sh -boot -R -B -V boostrap
```

and build *cabal* files (with some additional dependencies, flags for testing, and docs)

```
./build.sh -boot -R -B -V --moredeps --morecode --docs --discover --roundtrip cb
```

do a dry run (we are lazy and keep the flags, not needed really, target *dry* is short for `--dry install`)

```
./build.sh -boot -R -B -V --moredeps --morecode --docs --discover --roundtrip dry
```

(if happy with our dry run:) install

```
./build.sh -boot -R -B -V --moredeps --morecode --docs --discover --roundtrip install
```

run our test suite (cf. #testing below)

```
./build.sh -B -V runalltests
```

build our haddocks (copied to directory `html`)

```
./build.sh -B -V hd
```

build our *orgmode*/*html*/*pdf* docs (`docs/html`, `docs/pdf`)

```
./build.sh -B -V html
./build.sh -B -V pdf
```

etc.

testing

this section needs updating (in particular the old part below), short summary: the recommended way is to use the different test suites, relying on our *shake* tool chain (*runutests* for our *unit tests*):

```
$ ./build.sh -B -V runutests
```

```
$ ./build.sh -B -V runfiletests
```

```
$ ./build.sh -B -V rundoctests
```

```
$ ./build.sh -B -V runroundtrip
```

```
$ ./build.sh -B -V runtestsuite
```

or all of them together with:

```
$ ./build.sh -B -V runalltests
```

these require additional dependencies, and flags to be set, however (*runtestsuite* eg. requires the executable *discover*, which is not built with *Pire*'s core installation, etc.).

these dependencies can be ensured (flags can be set) by providing additional flags (at cabal file creation):

```
$ ./build.sh -B -V --discover cb
```

maybe use

```
$ ./build.sh -B -V --moredeps --discover cb
```

for even more dependencies, and recursively if you wish:

```
$ ./build.sh -R -B -V --moredeps --discover cb
```

this will set the *discover* flag then, and thus build *discover* as well.

> Tip
>
> we try to keep dependencies to a minimum (and want *Pire*'s core installation to be fast):
>
> strictly speaking the additional dependencies above (`--moredeps` etc.) are not needed, if you rely on *cabal*'s testing features (*Pire*'s #standard_installation):
>
> - `--enable-tests` (activated with `./build.sh --tests` as well),
> - passing flags to cabal: `-f "discover"` eg.,
> - and `cabal test`
>
> ie. create a cabal file without any additional dependencies as the default *pire.cabal*:
>
> ```
> ./build.sh -B -V cb
> ```
>
> and install with
>
> ```
> ./build.sh -boot -R -B -V --tests --doctests --roundtrip  --discover
>  install
> ```

(maybe add `--dry-run` for a dry run).

this calls:

```
cabal install --enable-tests -f "roundtrip" -f "discover" -f ...
```

he downside is that all the tests are then run at installation time (which can be slow) …

To test the old syntax modules (*OldSyntax*, *OldParser*, *OldSmart*, *OldNavigation*, etc.), cf. section #oldsyntax:

run the test suites above with `--oldsyntax`, this works currently only for the *unit tests*, and the *testsuite*:

```
$ ./build.sh -B -V --oldsyntax runutests
```

```
$ ./build.sh -B -V --oldsyntax runtestsuite
```

but then it shouldn't hurt to run the complete testsuite with `--oldsyntax` (even though not all of our tests take the *old syntax* into account yet).

```
$ ./build.sh -B -V --oldsyntax runalltests
```

the individual tests, one by one

Here is a more detailed list of our individual tests (yet to be updated):

1. unit tests: #utests
2. file tests: #filetests
3. doctests
4. roundtrip test

unit tests

```
$ cabal configure --enable-tests
$ cabal repl utest
```

load them with either

```
>>> :l tests/Tests.hs
```

or

```
>>> :l Tests
```

and run them

```
>>> main' "*"
```

or just:

```
>>> main
```

file tests

likewise for the file tests, eg.:

```
>>> :l tests/FileTests.hs
>>> main' "*"
```

Pi-forall in *Pire*

As mentioned already, *Pire* includes *Pi-forall* by means of two sub-projects (sub-directories):

- *pi-forall* is the original *Pi-forall* source code (the *full* distribution)
- *pire-pi-forall* makes *Pi-forall* available then in the *PiForall* name space,

It is possible then for example to use *Pi-forall*'s parser, and main program *goFilename*:

```
$ cabal new-repl pire-pi-forall
```

```
>>> :m +PiForall.Parser
>>> parseExpr   "   \\x  .  a  "
Right (Pos "<interactive>" (line 1, column 4) (Lam (<(x,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 11) (Var a))))

>>> :m +PiForall.Main
>>> goFilename "../pitests/Nat.pi"
...
```

etc.

export more parsers

old docs, yet to be updated/refined:

One thing that was useful in *Pire*'s old installation and should therefore be rescued (as it is not provided out of the box by the above approach): make more of *Pi-forall*'s parsers available: *PiForall.Parser* exports just *parseExpr*, but for more fine grained experimentation/testing one may want to test *Pi-forall*'s individual parsers as well: *expr*, *decl* etc., in addition to the commands above one can then test for example (the *expr* parser here):

```
$ cabal new-repl pire-pi-forall
```

```
>>> runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v}) []
 "<interactive>" " Succ n") emptyConstructorNames)
...
```

The Trifecta parsers: a tour

old docs, yet to be updated

Start the *cabal repl* and load the `_pire.hs` file (just a laundry list of further imports, for convenience), the system answers with a long list of modules being loaded (shortened here):

```
$ cabal new-repl
...
Loaded GHCi configuration from /home/rx/cfg/hs/dot-ghci
[ 1 of 51] Compiling Hint             ( Hint.hs, interpreted )
[ 2 of 51] Compiling OldSyntax        ( OldSyntax.hs, interpreted )
[ 3 of 51] Compiling OldSmart         ( OldSmart.hs, interpreted )
[ 4 of 51] Compiling OldNoPos         ( OldNoPos.hs, interpreted )
```

```
[ 5 of 51] Compiling OldNavigation    ( OldNavigation.hs, interpreted )
[ 6 of 51] Compiling OldForget        ( OldForget.hs, interpreted )
[ 7 of 51] Compiling OldText2String   ( OldText2String.hs, interpreted )
[ 8 of 51] Compiling PrettyCore       ( PrettyCore.hs, interpreted )
[ 9 of 51] Compiling PrettyCommon     ( PrettyCommon.hs, interpreted )
[10 of 51] Compiling Refactor         ( Refactor.hs, interpreted )
[11 of 51] Compiling Size             ( Size.hs, interpreted )
[12 of 51] Compiling Syntax           ( Syntax.hs, interpreted )
...
```

(or `cabal new-repl pire` ...)

```
>>> :l _pire
...
[ 1 of 52] Compiling Hint             ( Hint.hs, interpreted )
[ 2 of 52] Compiling PrettyCore       ( PrettyCore.hs, interpreted )
[ 3 of 52] Compiling PrettyCommon     ( PrettyCommon.hs, interpreted )
[ 4 of 52] Compiling Refactor         ( Refactor.hs, interpreted )
[ 5 of 52] Compiling Size             ( Size.hs, interpreted )
[ 6 of 52] Compiling Syntax           ( Syntax.hs, interpreted )
[ 7 of 52] Compiling Smart            ( Smart.hs, interpreted )
[ 8 of 52] Compiling Pretty2          ( Pretty2.hs, interpreted )
[ 9 of 52] Compiling Desugar          ( Desugar.hs, interpreted )
[10 of 52] Compiling TC               ( TC.hs, interpreted )
[11 of 52] Compiling TT               ( TT.hs, interpreted )
[12 of 52] Compiling Term             ( Term.hs, interpreted
...
>>>
```

Parse an identifier:

```
>>> parse (whiteSpace *> ide) "  wow this rocks"
"wow"
```

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

```
>>> parse (whiteSpace *> id2_) "  wow this rocks"
("wow",Pair (Id "wow") (Ws "  "))
```

Naming conventions are as follows:

- Parsers for Pi-forall's abstract syntax (absy parsers) have names like the original Pi-forall/Parsec ones: *identifier*, *reserved*, *expr* etc. (but sometimes their names have been shortened: *ide* for *identifier*, *var* for *variable* etc), and they parse the data structures of `Syntax.hs`: *Pire*'s flavour of *Pi-forall*'s syntax. Their code is in most cases similar (if not identical) to the original Parsec ones.
- Underscored versions of those names: *ide_* , *expr_* etc. are used for the corresponding concrete syntax parsers (again: simplified/shortened at times, especially for the ones used often: the building blocks).
- Any other variation of a name, like double primes: *id''* denote helper functions, experimental versions etc.

old syntax

In the *old syntax* both *expr*, and *expr_* parse the same data structures: expressions (*Expr*), likewise do *decl* and *decl_* parse declarations (*Decl*) etc., in *Expr* can then pairs of constructors be found,

- eg. *If* for *if-then-else*,
- and its white space aware counterpart: *If_*.

More complicated expressions can be parsed, too:

```
>>> nopos $ parse (whiteSpace *> expr) "   \\n . if nat_leq n 0 then 1 else mult n (fac
 (minus n 1))  "
LamPAs [(RuntimeP,"n",Annot Nothing)] (Scope (If ((V (F (V "nat_leq")) :@ V (B 0)) :@ Nat
 0) (Nat 1) ((V (F (V "mult")) :@ V (B 0)) :@ Paren (V (F (V "fac")) :@ Paren ((V (F (V
 "minus")) :@ V (B 0)) :@ Nat 1))) (Annot Nothing)))
```

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

```
>>> nopos $ parse (whiteSpace *> expr_) "   \\n . if nat_leq n 0 then 1 else mult n (fac
 (minus n 1))  "
LamPAs_ (LamTok "\\" (Ws "")) [(RuntimeP,Binder_ "n" (Ws " "),Annot Nothing)] (Dot
 "." (Ws " ")) (Scope (If_ (IfTok "if" (Ws " ")) ((Ws_ (V (F (V "nat_leq"))) (Ws "
 ") :@ Ws_ (V (B 0)) (Ws " ")) :@ Nat_ 0 "0" (Ws " ")) (ThenTok "then" (Ws " ")) (Nat_
 1 "1" (Ws " ")) (ElseTok "else" (Ws " ")) ((Ws_ (V (F (V "mult"))) (Ws " ") :@ Ws_ (V
 (B 0)) (Ws " ")) :@ Paren_ (ParenOpen "(" (Ws "")) (Ws_ (V (F (V "fac"))) (Ws " ") :@
 Paren_ (ParenOpen "(" (Ws "")) ((Ws_ (V (F (V "minus"))) (Ws " ") :@ Ws_ (V (B 0)) (Ws "
 ")) :@ Nat_ 1 "1" (Ws "")) (ParenClose ")" (Ws ""))) (ParenClose ")" (Ws "  "))) (Annot
 Nothing)))
```

Of course one wants some kind of pretty printing here

```
>>> pp $ fromSuccess $ parseString (runInnerParser $ evalStateT (whiteSpace *> expr)
 piInit) beginning " \\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))))
```

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:

```
>>> pp $ nopos $ parse (whiteSpace *> expr) "   \\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))
```

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`):

(new syntax:)

```
>>> pp $ fst $ parse expr_ "\\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))


>>> pp $ snd $ parse expr_ "\\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))
```

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: some examples can be found in the unit tests section (in the `tests` directory), and in the doctests throughout the program code (in the `src` directory).
- 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.

- explain *desugaring* (conversion of lambdas, Nats)
- explain the *forget* function
- explain how *layout parsing* is handled.
- explain *untie*

The Zipper implementation

As of November 2018 the zipper implementation has changed completely, this section thus needs updating.

(old docs as of August 2015, describing changes in our zipper implementation at the time): the basic idea is this:

The original zipper implementation (closely following the Zippers chapter [http://learnyouahaskell.com/zippers] of *Learn you a Haskell*) uses a list of breadcrumbs like so:

```
type Zipper a = (a, [Crumbs a])
```

the first part of this pair being the current focus, and the crumbs (or context) a means to get up in the tree, or reconstruct the original tree.

Thus, eg. in an apply, if we take a left turn:

```
left (Exp (lhs :@ rhs), bs) = rsucceed (Exp lhs, (AppLeftCrumb $ rhs) : bs)
```

we return the left hand side as the new focus, and keep track of the fact that we have taken a left turn, together with the right hand side in the *AppLeftCrumb* (the *Exp* is just a means to cope with the fact, that our tree has exprs/decls/modules etc.)

When going up, we have to handle those breadcrumbs

```
up (Exp e, AppLeftCrumb rhs : bs) = rsucceed (Exp $ e :@ rhs, bs)
```

This works so well, but obviously there are many different situations to take into account, and therefore many different constructors for breadcrumbs.

I have changed the breadcrumbs now to be a list of functions instead

```
type Zipper a = (Tr a, [Tr a -> Tr a])
```

(where *Tr a* is the tree of exprs, decls or modules). Thus to take a left turn, we add to the list of functions just another one:

```
left (Exp (l :@ r), bs) = rsucceed (Exp l, (\(Exp l') -> Exp $ l' :@ r) : bs)
```

This functions serves the same purpose: tells us how to get back, no need to handle any data structures for breadcrumbs though, resulting thus in much shorter code.

Another advantage is that the left turn and the function how to get back up are visually in the same place (whereas formerly they where in two different functions: *left* and *up*). *up* is trivial now: just an application.

The only caveat I would say is that one cannot easily look at these functions as breadcrumbs (but I can imagine even implementing some *Show* instance for these *Tr a # Tr a* functions at some point).

I owe this idea of using functions to Darryl McAdams, author of the From Zipper to Lens [https://www.fpcomplete.com/user/psygnisfive/from-zipper-to-lens] tutorial. This is a nice tutorial (meant as a motivation for Lens), but I would claim, that he is not implementing zippers there: zippers do allow to climb up the tree in a step by step manner, and we need that, for *upToBinding* / *upToBinding'* at least (and therefore keep *a list of functions*). With just a single function kept in his zippers though, one can only get back straight to the top of the tree.

Pure lenses are not suited for zippers, for this very reason, as I understand, but there are of course other zipper implementation, one of them being based on lenses: Control.Zipper [https://hackage.haskell.org/package/zippers-0.2/docs/Control-Zipper.html]

I am doing some experimentation with them currently, likewise with *Lens.Action*, and the code has therefore some some additional dependencies on *lens-action* and *zippers* currently. They are not strictly necessary, and may be removed in the future again.

(Another generic approach would be *scrap you zippers*, maybe worth looking into).
Anyway, for the time being, I have decided to stick with our simple hand written zippers, and have stolen just this one idea: to use functions (lists of functions ie.) as breadcrumbs. Pi-forall being not that huge, fortunately, this approach is still feasible. I am glad, that the zipper implementation gets simpler this way (and I can imagine that it could be shortened even more, with *Control.Zipper* eg.).

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.

Moreover, 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 worth 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 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:

```
>>> import PiForall.Main
>>> goFilename "samples/Sample.pi"
processing Sample.pi...
Parsing File "samples/Nat.pi"
Parsing File "samples/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))
```

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.

> Tip
>
> The output of some of the following examples is very long, and has thus been shortened — not just by hand, but on the *ghci* (*cabal repl*) command line as well, to use them as doctests nevertheless. Having imported some modules initially:
>
> ```
> import PiForall.Modules
> import PiForall.PrettyPrint
> import Control.Monad.Except
> import Data.Either.Combinators
> ```
>
> we'd just write
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi")
> ```
>
> eg., and see a list of modules being parsed, many lines of output ie., and we'd see the same list of modules, if we just return them after parsing:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= return
> ```
>
> We can get the result of the *Either* monad with *fromRight'* from the either package:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= return . fromRight'
> ```
>
> and we'd see the same output, if we *show* that list of modules and print the resulting *String*:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . show .
>  fromRight'
> ```

even though, technically, we are looking at the *String* representation of the modules here. Taking this one step further, we can shorten the output to the first 400 characters:

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((+
+"...") . take 400) . show . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
[Module {moduleName = "Nat", moduleImports = [], moduleEntries = [Data
 "Nat" Empty [ConstructorDef "samples/Nat.pi" (line 12, column 3) "Zero"
 Empty,ConstructorDef "samples/Nat.pi" (line 13, column 3) "Succ" (Cons
 Runtime _ (Pos "samples/Nat.pi" (line 13, column 12) (TCon "Nat" []))
 Empty)],Sig is_zero (Pos "samples/Nat.pi" (line 15, column 11) (Pi (<(_1,
{TCon "Nat" []})> TyBool))),Def is_zero (Po...
```

Adjust the following examples to your needs in this manner.

Assuming the module imports above, ie. initially just:

```
import PiForall.Modules
import PiForall.PrettyPrint
import Control.Monad.Except
import Data.Either.Combinators
```

you will need more imports in the course of the following examples (import them now, or later when needed):

```
import PiForall.Syntax
import PiForall.Parser
-- import Unbound.LocallyNameless
import Unbound.Generics.LocallyNameless
import Control.Monad.State.Lazy
import Text.Parsec
import qualified Data.Set as S
```

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.Except* monad, like so:

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((++"...") . take
 400) . show . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
[Module {moduleName = "Nat", moduleImports = [], moduleEntries = [Data "Nat" Empty
 [ConstructorDef "samples/Nat.pi" (line 12, column 3) "Zero" Empty,ConstructorDef
 "samples/Nat.pi" (line 13, column 3) "Succ" (Cons Runtime _ (Pos "samples/Nat.pi" (line
 13, column 12) (TCon "Nat" [])) Empty)],Sig is_zero (Pos "samples/Nat.pi" (line 15,
 column 11) (Pi (<(_1,{TCon "Nat" []})> TyBool))),Def is_zero (Po...
```

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. here, and again, only the beginning of the output is shown):

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((++"...") . take
 400) . show . last . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
Module {moduleName = "Fac", moduleImports = [ModuleImport "Nat"], moduleEntries =
 [Sig two (Pos "samples/Sample.pi" (line 13, column 7) (TCon "Nat" [])),Def two (Pos
```

```
 "samples/Sample.pi" (line 14, column 7) (DCon "Succ" [Arg Runtime (Paren (Pos "samples/
Sample.pi" (line 14, column 13) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot
 Nothing))] (Annot Nothing))))] (Annot Nothing))),Sig nat_leq (Pos ...
```

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

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= return . disp . last . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/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))))
```

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 Simple.pi:

```
-- Simple.pi


-- fromSuccess  `liftM` parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many
 decl) piInit) "samples/Simple.pi"
-- resp
-- parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many decl) piInit)
 "samples/Simple.pi" >>= return . fromSuccess
-- parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many decl) piInit)
 "samples/Simple.pi" >>= return . nopos . fromSuccess



module Fac where


-- caseExpr w/ prelude
-- case x of
--    Zero -> True
--    Succ m -> False


-- caseExpr w/ prelude
-- case y of
--     Zero -> False
--     Succ n -> nat_leq m n
```

```
-- caseExpr w/ prelude
-- case x of
--    Zero -> True
--    Succ m -> case y of
--           Zero -> False
--           Succ n -> nat_leq m n


-- lambdaPAs w/ prelude
-- \ x y .
--   case x of
--      Zero -> True
--      Succ m -> case y of
--             Zero -> False
--             Succ n -> nat_leq m n



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):

```
>>> runExceptT (parseModuleFile  emptyConstructorNames  "samples/Simple.pi") >>= return .
 disp . fromRight'
Parsing File "samples/Simple.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))))
```

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

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

Again, pretty printing is convenient for reading the result:

```
>>> disp $ fromRight' $ parseExpr   "   \\x  .  a  "
\x . a
```

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 (the import as above, assumed already): ie. we can parse expressions, declarations, signature definitions, variables etc. (shown is parsing an expression):

```
>>> 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)))
```

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 recognising *Succ* as a data constructor rather than as a regular function:

```
>>> let prelude = ConstructorNames (S.fromList ["Nat"]) (S.fromList ["Zero", "Succ"])
>>> 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)))
```

Backtracking in parsers and *Pi-forall* mimicry

> This chapter describes some implementation details: how *Pire*'s *Trifacta* parser behaved differently from *Pi-forall*'s *Parsec* parser originally, and how it mimics *Pi-forall* / *Parsec*'s exact behaviour now.
>
> These details may not be essential for understanding *Pire*'s approach to refactorings, but maybe they are interesting as a bug hunting experience (if one may call *Pire* / *Trifecta*'s original behaviour a bug), and they should give some insight, what's happening under the hood.

We have solved the problem of parsing `A->BA->B` (and similar more complicated cases, cf. the examples below) in exactly the same manner as *Pi-forall* does.

To make a long story short: All of *Pi-forall*'s test files parse fine now (comparing the result of `untie $ parse …` with *Pi-forall*'s parser), as covered by the tests.

tl;dr

Strictly speaking: yes, maybe it's not that important, if the fresh variables used read as

> `_, _1, _2, _3, …` - as always in *Pire*'s *Trifecta* parser's case (so far)

or as in *Pi-forall* / *Parsec*'s case:

- sometimes as `_, _1, _2, _3, …`
- at other times as `_2, _3, _4, …`
- or `_4, _5, _6, …`
- but also as `_16, _17, _18, …`
- etc.

depending on the level of parentheses / number of arrows etc.

Nevertheless, we feel much more comfortable now, that we have found a way to make our *Trifecta* parser behave exactly in *Pi-forall*'s manner (Pi-forall *mimicry*), and understand what's going on.

We can thus continue to be very strict in our comparisons of results: *parse* & *untie* vs. *Pi-forall*'s original parser, and would hope that this way, other possible subtle differences / future issues won't go unnoticed.

A few examples may be worth looking at (and yes, there is a system to this, it may just not be obvious in the first place, how it works `:-)`):

```
> import PiForall.Parser as P
> import Pire.NoPos
> import Data.Either.Combinators (fromRight')

> nopos $ fromRight' $ P.parseExpr "A->B"
Pi (<(_,{Var A})> Var B)

> nopos $ fromRight' $ P.parseExpr "A->B->C"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Var C))

> nopos $ fromRight' $ P.parseExpr "A->B->C->D"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Var D)))

> nopos $ fromRight' $ P.parseExpr "A->B->C->D->E"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Pi (<(_3,{Var D})> Var E))))

> nopos $ fromRight' $ P.parseExpr "(A->B)"
Paren (Pi (<(_,{Var A})> Var B))

> nopos $ fromRight' $ P.parseExpr "(A->B->C)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Var C)))

> nopos $ fromRight' $ P.parseExpr "(A->B->C->D)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Var D))))

> nopos $ fromRight' $ P.parseExpr "(A->B->C->D->E)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Pi (<(_3,{Var D})> Var
 E)))))

> nopos $ fromRight' $ P.parseExpr "((A->B))"
Paren (Paren (Pi (<(_2,{Var A})> Var B)))

> nopos $ fromRight' $ P.parseExpr "((A->B->C))"
```

```
Paren (Paren (Pi (<(_4,{Var A})> Pi (<(_5,{Var B})> Var C))))

> nopos $ fromRight' $ P.parseExpr "((A->B->C->D))"
Paren (Paren (Pi (<(_6,{Var A})> Pi (<(_7,{Var B})> Pi (<(_8,{Var C})> Var D)))))

> nopos $ fromRight' $ P.parseExpr "((A->B->C->D->E))"
Paren (Paren (Pi (<(_8,{Var A})> Pi (<(_9,{Var B})> Pi (<(_10,{Var C})> Pi (<(_11,{Var
 D})> Var E))))))

> nopos $ fromRight' $ P.parseExpr "(((A->B)))"
Paren (Paren (Paren (Pi (<(_8,{Var A})> Var B))))

> nopos $ fromRight' $ P.parseExpr "(((A->B->C)))"
Paren (Paren (Paren (Pi (<(_16,{Var A})> Pi (<(_17,{Var B})> Var C)))))

> nopos $ fromRight' $ P.parseExpr "(((A->B->C->D)))"
Paren (Paren (Paren (Pi (<(_24,{Var A})> Pi (<(_25,{Var B})> Pi (<(_26,{Var C})> Var
 D))))))

> nopos $ fromRight' $ P.parseExpr "(((A->B->C->D->E)))"
Paren (Paren (Paren (Pi (<(_32,{Var A})> Pi (<(_33,{Var B})> Pi
(<(_34,{Var C})> Pi (<(_35,{Var D})> Var E)))))))

> nopos $ fromRight' $ P.parseExpr "((((A->B))))"
Paren (Paren (Paren (Paren (Pi (<(_26,{Var A})> Var B)))))

> nopos $ fromRight' $ P.parseExpr "((((A->B->C))))"
Paren (Paren (Paren (Paren (Pi (<(_52,{Var A})> Pi (<(_53,{Var B})> Var C))))))

> nopos $ fromRight' $ P.parseExpr "((((A->B->C->D))))"
Paren (Paren (Paren (Paren (Pi (<(_78,{Var A})> Pi (<(_79,{Var B})> Pi (<(_80,{Var C})>
 Var D)))))))
```

etc.

We started out reducing *Pire*'s parser and later *Pi-forall*'s parser to the bare minimum of functions that would still show the above behaviour (or lack thereof in *Pire*'s case initially), i.e. got rid of *declarations*, and lots more stuff.

Then we added some writing capabilities to *Pire*'s parser: ran it in the *RWS (Reader-Writer-State)* monad, rather than just in the *State* monad, which allowed us sprinkle some *tell me*'s in the code, and also we added some increment function calls here and there (that would increment the state's counter for fresh variables).

We expirmented with commenting out some of the code, eg. choices of the grammar as well.

That way we found that one particular *choice* function within the local function definiton of *beforeBinder* in *expProdOrAnnotOrParens* was at the culprit, originally:

```
beforeBinder = parens $
  choice [do e1 <- try (term >>= (\e1 -> colon >> return e1))
             e2 <- expr
             return $ Colon' e1 e2
         , do e1 <- try (term >>= (\e1 -> comma >> return e1))
              e2 <- expr
              return $ Comma' e1 e2
         , Nope <$> expr]
```

We could change *Pire*'s parsing behaviour by adding some *increments* and *tell me*'s (it just wouldn't behave like *Pi-forall*'s parser still), like so:

```
choice [
  do
    tell "in colon branch\n"
    e1 <- try (term >>= (\e1 -> colon >> return e1))
    e2 <- expr
    return $ Colon' e1 e2
, do
   tell "in comma branch\n"
   e1 <- try (term >>= (\e1 -> comma >> return e1))
   e2 <- expr
   return $ Comma' e1 e2
, do
    tell "in comma branch\n"
    increment
   , Nope <$> expr
]
```

Likewise commenting out for example the middle comma choice above (and in *Pi-forall*'s parser) had an impact in terms of fresh variables' choice.

We were still far from real *Pi-forall* mimicry, no matter what we tried: where we put our increment function calls etc.: we couldn't make *Pire*'s parser behave in *Pi-forall*'s way. But at least that got us on the right track for understanding what's going on:

Pi-forall's parser is defined as:

```
type LParser a = ParsecT
                   String                      -- The input is a sequence of Char
                   [Column] (                  -- The internal state for Layout tabs
                      StateT ConstructorNames
                      FreshM)                  -- The internal state for generating
 fresh names,
                   a                           -- the type of the object being parsed
```

Never mind the details, just notice that that the monad transformer stack is:

> *ParsecT(…(StateT…(FreshM))*

with *FreshM* within *ParsecT*. It is possible to use Parsec the other way around (*Parsec* inside *StateT*)

> *StateT… Parsec*

it's just not done that way in *Pi-forall*, and the two behave differently:

> the *StateT … Parsec* does backtracking for the state in case a choice fails,
>
> the *ParsecT(…(StateT…(FreshM))* does not do (cannot do) any backtracking, cf. for example Roman Cheplyaka' blog: https://ro-che.info/articles/2012-01-02-composing-monads

The *Trifecta*-parser is written in this later style:

> *StateT …(inner trifecta parser)*

and does backtracking. We were happy with the parser so far, (this is the style of the ' parser), and if we wouldn't want to mimic *Pi-forall*'s behaviour, our choice of fresh vars seemed more natural.

Trifecta parsers cannot be written in a monad transformer (*TrifectaT*) style (not that we are aware of: it is not in the documentation, we have never seen a *Trifecta* parser as a monad transformer wrapping a state, it's always the state wrapping the *Trifecta* parser).

A reduced to the bare minimum *Pi-forall* grammar shows what is going on:

```
expr ->
        -- pi-type, right-associativity with the help of buildExpressionParser table
        term "->" term

        | term

term ->
        -- App
        factor @: factor

        | factor

factor -> var

          -- annotation Ann
          | "(" term ":" expr ")"

          -- Prod
          | "(" term "," expr ")"

          -- Paren
          | "(" expr ")"
```

Now see, what happens when we parse `A -> BA -> B` as an *expr*: well, we must find a *term*, and in turn a *factor* then:

Oh, it starts with `(`, cannot just be a *var* then, but must be one of the other cases, so try them (in this order):

```
| "(" term ":" expr ")"
| "(" term "," expr ")"
| "(" expr ")"
```

so have a look what's inside the outer parentheses: `(A->B)`, and try to parse that as a *term*: well that's possible: `(A->B)` is a term (therby using some fresh variables for the *Π*-type found), it is just not followed by a `:`, so put back.

> in the *Trifecta* case: backtracking clears the state as if we had never used any fresh vars for the *Π*-type found
>
> in *Pi-forall*'s case: the fresh vars for the term / *Π*-type `(A->B)` are used forever.

try next possibility, similar: parsing `(A->B)` as a term succeeds (und renders fresh variables used in the case of *Pi-forall*), just not followed by `,`

so finally go for the last case: `(` *expr* `)`.

This also explains, why longer arrow chains `A->B->C->DA->B->C->D` make a difference: use `_, _1, _2` in the first attempt of parsing the inner `(A->B->C->D)`, `_3, _4, _5` in the second attempt, and finally return `_6, _7, _8`

So how could we mimic *Pi-forall*'s behaviour?

Well the solution was to rely on *lookAhead*:

```
choice [do { e1 <- try (term >>= (\e1 -> colon >> return e1)) ;
             e2 <- expr ;
             return $ Colon' e1 e2
           }
        , do { lookAhead term ;
               e1 <- try (term >>= (\e1 -> comma >> return e1)) ;
               e2 <- expr ;
               return $ Comma' e1 e2
             }
         , do { lookAhead term ;
                lookAhead term ;
                Nope <$> expr
              }
         ]
```

If we are in the second (comma) case, we must already have tried the first case (and succeeded with term): *lookAhead* doesn't consume the input, put triggers the state's counter increment (as if no backtracking had happened)

Likewise: if we are in the third case: we must have looked at the other two case already (and found a term each time): thus call *lookAhead* twice.
OK, we are very glad, we got this sorted out, and will continue with the finishing touches for a new Pire release, soon, and turn my attention to the refactorings.

Changes

complete rewrite as of December 2017, summary of major changes and some design decisions (ordered by topic), recent and older ones:

- Pire is based on Edward Kmett's Bound library instead of Unbound, used in Pi-forall
- While an older version of Pire included Pi-forall as a library, the current one does not.

  It was used for testing Pire's parser against Pi-forall's original parser at the time, and will possibly be reintroduced again.

  Thus functions to translate between the various syntax representations, like `untie` have disappeared as well for now.

concrete syntax and white space aware parsing, naming conventions

- For every abstract syntax expression (`If`, say), a white space aware version (`If_`) was given in the past, covering not only the abstract syntax essentials, but also any white in between. Necessarily these expressions (declarations, modules etc.) were complicated:

  While `If` was defined as

  ```
  If (Expr t a) (Expr t a) (Expr t a) (Annot t a)
  ```

  eg., its white space aware counterpart `If_` was

  ```
  If_ (Token 'IfTokTy t) (Expr t a) (Token 'ThenTokTy t) (Expr t a) (Token 'ElseTokTy t)
   (Expr t a) (Annot t a)
  ```

each of the token and subexpressions allowing for trailing white space.

`expr_` was the parser of these white space aware expressions (and still is), `expr` being the ordinary abstract syntax tree parser.

This naming convention applies to module names as well: `Modules_` eg. is the white space aware version of `Modules`, with some exceptions: `WsAwareParser` vs. `Parser` notably.

These white space aware expression (like `If_` above) turned out

- likewise nopos (no position) operates on Absy_Bound (but on ConcreteSyntax_Bound and Absy_Unbound as well with the help of type classes): get rid of position information to make the parsers output more human readable

testing

- tasty/hunit test available in Pire/Tests.hs and Pire/FileTests.hs There are currently 340 tests in Pire/Tests.hs and 46 tests in FileTests.hs to provide compatibility with Pi-forall

- doctests in most of the source files, run them with

```
$ doctest -isrc src/Pire/Parser/Basic.hs
$ doctest -isrc Pire.Parser.Basic
$ doctest -isrc src/Pire/Parser/*
```

  eg.

- The unit tests have been moved from the src directory to their own tests directory, and are treated as a (cabal) test-suite utests, use them eg. with:

```
$ cabal configure --enable-tests
$ cabal repl utests
```

```
>>> :l tests/Tests.hs
>>> main' "*"
>>> :l tests/FileTests.hs
>>> main' "*"
```

- testing has been improved: the doctests, while in place for a while already, usage like this:

```
$ find ./src -iname "*.hs" | egrep -v -i "keepme|whatever" | xargs doctest -isrc
```

  can now be called from a separate `doctests.hs` program in the tests subdir, and are part of the the cabal test-suite, they can be called togehter with:

```
$ cabal configure --enable-tests
$ cabal build
$ cabal test
```

  Three tests currently: unit tests (utests), file tests (filetests), and doctests. OK, more docs to come, how to use them, but these should go in the docs section, not the changes section.

- a kind of literate programming for the docs introduced (with special `// doctest` … `// /doctest` comments, more details in the docs/README [https://github.com/reuleaux/pire/blob/master/docs/

README.adoc#doctest_snippets], ie. the other way around as extracting code snippets, as recently introduced), the idea is to have readable adoc files, with examples doctested nevertheless, cf. eg. the Parsec chapter [parsec.xml#parsec_parsers]

- The individual tests in Tests.hs pointing to precise features/requirements each, were easier to handle in case of failures, than the relatively complex file tests in FileTests.hs.

  Thus more individual tests were added: there are 468 of them now (plus the 57 file tests).

t2s is handy when working on the command line

- Working on the command line it seems most natural to work with Strings, like so

```
>>> V "a" :@ V "b"
...
>>> lam "x" $ V "a" :@ V "b"
...
>>> Ws_ (V "b") $ Ws "{-foo-}"
...
```

  These are examples of String expressions

```
>>> :t Ws_ (V "b") $ Ws "{-foo-}"
Ws_ (V "b") $ Ws "{-foo-}" :: Expr [Char] [Char]
```

  but the parser yields Text expressions, eg.

```
>>> nopos $ parse lambda " \\ x . a"
Lam "x" (Scope (V (F (V "a"))))
>>> :t nopos $ parse lambda " \\ x . a"
nopos $ parse lambda " \\ x . a" :: Pire.Syntax.Ex.Ex
```

```
>>> t2s $ nopos $ parse lambda " \\ x . a"
Lam "x" (Scope (V (F (V "a"))))
>>> :t t2s $ nopos $ parse lambda " \\ x . a"
t2s $ nopos $ parse lambda " \\ x . a" :: Expr String String
```

  Yes, one *can* use `:set -XOverloadedStrings` on the ghci (cabal repl) command prompt, but this only so good: often enough, it's not clear what is meant by `"foo"` then: the String, the Text?

- This t2s conversion one wants to do with the least effort of course, and cheating at the *ermine* source code (as for many things *Bound* related as well), *bitraverse* (*Bitraversable* etc) comes in handy: once expressed, how to bitraverse the expression tree in general, the String to Text conversion of Expr t a becomes just a bitraversal with T.unpack for both: its t and a parameters.

position decoration…

not currently implemented…

- can use bitraversal to decorate the syntax tree with (very fine grained) position information: ie. for every identifier, token, white space etc, all this after parsing. (independent of the (Pos …) info wrapped around some exprs by the parser: didn't want to completely change the parser again, and found this position info used too sparingly)

  had to change tokens for this to work: every token keeps track of the string (text) parsed, in order to participate in this decoration with position info

likewise had to reintroduce white space aware constructors for Type_, TyBool_ etc, had removed them just recently (oh well!), because I thought I could save them with Ws_ (Type t) Ws eg., but now we need not only the white space after the Type, but also the string (text) "Type" itself, thus Type_ t (Ws t), ie. the t in the middle.

anyway, happy with the position decoration, watch out for bound names, though (some examples in Decorate.hs). This will hopefully not be a problem as we need to open binders (instantiate lambdas etc), as we navigate the tree.

beginning navigation in the concrete syntax tree (parsed w/ white space), more to come, likewise more explanations, examples.

- *uwrap* introduced in addition to *wrap*. *unwrap* can be simpler (rely on hoistScope ie.), recall: wrap allows bound vars to be wrapped w/ their textual representation, for they can contribute to the calculation of the position info)

- further groundwork for navigation in the white space aware syntax tree, ie. functions/classes/ general approuch in `Decorate.hs` and `Refactor.hs` refined, to be continued.

- position info, ranges, and navigation in the ws aware syntax tree, and refactoring split into pieces and moved to Refactor/*.hs

- Text2String supports modules now

- can use bifoldMap/foldMap on the decorated syntax tree (with position info ie.) to calculate ranges (similar in spirit to bitraversal), range calculation thus simplified

- first steps of walking the (ws aware) tree by line column number

- managed to even traverse ConstructorNames (use them as an applicative), by means of a poor man's traversal on sets. decorateM thus returns a module now (the old tuple version is still available as decorateM'), decorated modules can thus be treated as any other pieces of the zipper

- *Decorated* typeclass and instances introduced, and constraints for /Range/s simplified that way.

pretty printing

- made pretty printing work on T.Text only, to simplify matters, but provided printing of String Exprs/ Decls/Modules etc by means of Text2String.

simple exprs

- white space aware versions of the SimpleExpr parsers introduced as well: `simple_expr`, `simple_term` etc

- the Expr constructor Pos renamed to Position, so Pos could be used for the constructor of position info as calculated / decorated (formerly LineColumn), just this one constructor now (ie. From, To, AnyPos merged to Pos)

described here are the steps how to build the (html, pdf) docs from their .adoc sources.

as mentioned already: rendered versions of these docs can be found on http://a-rx.info/pire

building the docs

- building both: the html and the pdf requires some code snippets to be extracted first from Haskell files (code snippets)
- and the the other way around as well: some of the example sessions in the adoc files, can be extracted into Haskell files and doctested (doctest snippets)
- building the pdf w/ docbook additinally requires some font metric files from their ttf files to be built (font metric files, rethink: maybe newer versions of fop can use the system fonts?),
- in any case *shake* needs to be installed first

  (shake is not strictly a requirement for Pire, only for the docs, thus can/should be removed as a dependency from the cabal file maybe?)
- all of the following build steps can be *undone* by

  ```
  ./build.sh -B -V clean
  ```

  which will remove the .pdf, .fo, .html etc files

building the code snippets

- work is underway to replace some of the code snippets and command line examples in the docs, with ones that have been properly doctested, thus they need to be created first:

  ```
  $ ./build.sh -B -V allsnippets
  ```

  in case you only want to build the html, proceed w/ building the html
- it is nice that github does .adoc rendering out of the box, but unfortunate that includes are turned into links (for security reasons, as I understand), and given that these snippets are generated from their haskell source files, they are not included in pire's git repo, but rather =.gitignore=d, and thus the docs are readable on github only so well. - I do care about the final html/pdf (as found on http://a-rx.info/pire eg., or built yourself following the instructions here), and about properly tested examples (rather than about easily readable on github docs, cf. eg. file:tour.xml [tour.xml])

doctest snippets

- in order to keep the .adoc files readable, some of the examples work the other way around: a haskell file `_parsec.hs` is created from the corresponding `parsec.adoc` eg. with

  ```
  $ ./build.sh -B -V _parsec.hs
  ```

  and all of them together in one go with

  ```
  $ ./build.sh -B -V writedoctests
  ```

  They will be copied to the src directory, where they will subsequently considered by the doctests

  Special `// doctest` … `// /doctest` comments are used in the adoc files (and likewise `// setup` … `// /setup` for the initial setup).

This is a kind of (self made / ad hoc) literate programming: Haskell doctests from adoc files, only for the purpose of having properly tested docs, however (ie. all the modules under src neither contain literate Haskell, nor are they created from adoc files: they use just plain Haskell)

building the html

- the html toolchain is simpler than the docbook toolchain in that it builds upon asciidoctor

```
./build.sh -B -V html
```

- but html can be created from the docbook as well, thus there is a .dbkhtml target in `Build.hs` (not up to date, however)

building the font metric files

- their creation is separate from the rest of the docbook tool chain: they are not =need=ed by shake, for I didn't want them to be rebuilt with every call of:

```
./build.sh -B -V pdf
```

(which I run quite often, maybe rethink: find a better way).

- thus first build the font metric files (*f* for fonts in *fcreate*)

```
./build.sh -B -V fcreate
```

and then proceed with building the pdf as described building the pdf

- `build fcreate` touches an (emtpy) file `fcreate`, indicating that the metric files have been created
- their creation can be *undone* w/

```
./build.sh -B -V fclean
```

(aka fonts clean) for the font metric files

building the pdf

- build the pdf w/

```
./build.sh -B -V pdf
```

(with or without -B, -V)

remarks

- the install dir lists some (debian) deps that were helpful for setting up the docbook tool chain in the past (todo: see if they are all still valid, clean up)
- the toolchain relies on some xsl style sheets that I have written in the past (they work for me), but could probably benefit from some cleanup: not all the intermediate steps / xsl transformations (.dbk + files etc) are really needed, I guess.

- the ttf fonts used are free ones (Palatino from http://www.webpagepublicity.com/free-fonts-p.html and LucidaTypewriter form http://www.fontsner.com/) and are included now in the fonts directory to make the the docbook toolchain self contained.
- the xslt*.rnc (RelaxNG grammar files, for keeping emacs happy, with by means of `schemas.xml`) are from Norman Walsh (https://github.com/ndw/xslt-relax-ng).