An Introduction to Haskell and Naproche-SAD

by Peter Koepke

September 2019

1 Introduction

The focus of this text is Haskell programming for the purposes of the Naproche (Natural Proof Checking) project. Haskell appears as a language which is difficult to learn, write and read. Instead of learning Haskell in isolation with toy examples or with powerful but nerdy 1-liners we immediately consider actual modules in the current Naproche-SAD system. This text is work in progress, comments and corrections are very welcome.

In these notes, sophisticated notions like parsers and monads are not motivated theoretically, but they are simply part of the programming style in the Naproche-SAD system. We shall consider modules in ascending order of difficulty, starting with rather straightforward code on line and column numbers as actually used by Naproche-SAD and the Isabelle/jedit editor. We shall then proceed to tokenization code and parsers. We shall often ignore technical parts of code like the all-pervading error handling (and also parts of the code that I do not understand). One can use combinations of monads and parsers even if one does not understand how, e.g., an instance of the operation `>>=` for parsers has been programmed. Just like we do not know how the operation $+$ for natural numbers is "really" implemented in our computers.

Our emphasis is on *parsing* in Naproche-SAD. This is also motivated by a project to translate ForTheL texts into the type-theoretic language Lean. We are aware that the present code of Naproche-SAD could be greatly improved by using sophisticated Haskell techniques and libraries, and this will eventually happen. But we think that it is helpful for beginners to work with rather tangible data types like integers and strings with low-order functions and slowly progress to highly polymorphic higher order functions.

These notes can be followed practically with any text editor and the interactive Haskell compiler GHCi in a Unix terminal. A later chapter called *Programming Naproche-SAD* also describes, how code development could also be done with the Isabelle-jedit user interface.

2 The ForTheL controlled natural language for mathematics

Our emphasis is on the natural language of mathematics and its processing. The common language of mathematics is a characteristic mixture of natural language phrases together with symbolic "formulas". Sometimes mathematical results are expressed just by words. Tom Hales' main theorem in *A proof of the Kepler conjecture* reads:

> Theorem 1.1 (The Kepler conjecture). No packing of congruent balls in Euclidean three space has density greater than that of the face-centered cubic packing.

A little further on, statements become more typical using combinations of natural language phrases and symbolic material:

> Lemma 1.3. If there exists a negligible fcc-compatible function $A:\Lambda\to\mathbb{R}$ for a saturated packing Λ, then there exists a constant C such that for all $r \geq 1$ and all $x\in\mathbb{R}^3$,
>
> $$\delta(x,r,\Lambda)\leq\pi/\sqrt{18}+C/r.$$
>
> The constant C depends on Λ only through the constant C_1.

Mathematicians can give precise formal meaning to these semi-natural statements. The language ForTheL (Formula Theory Language) of the Naproche-SAD project intends to approximate the semi-natural language of mathematics.

In ForTheL one can axiomatically introduce the notions involved in the formulation of Theorem 1.1:

```
[synonym number/-s]
Signature. A real number is a notion.
Let x,y stand for real numbers.

Signature. x is greater than y is an atom.

Signature. A packing of congruent balls
in Euclidean three space is a notion.

Signature. The face centered cubic packing is a packing
of congruent balls in Euclidean three space.

Let P denote a packing of congruent balls in
Euclidean three space.

Signature. The density of P is a real number.

Theorem The_Kepler_conjecture. No packing of congruent
balls in Euclidean three space has density greater than
the density of the face centered cubic packing.
```

This file, named `Hales.ftl`, is parsed successfully in Naproche-SAD but surely the verification fails:

```
[Parser] (file "/home/koepke/TEST/Hales.ftl")
parsing successful
[Reasoner] (file "/home/koepke/TEST/Hales.ftl")
verification started
[Translation] (line 2 of "/home/koepke/TEST/Hales.ftl")
forall v0 ((HeadTerm :: aRealNumber(v0)) implies truth)
[Translation] (line 5 of "/home/koepke/TEST/Hales.ftl")
(aRealNumber(x) and aRealNumber(y))
[Translation] (line 5 of "/home/koepke/TEST/Hales.ftl")
((HeadTerm :: isGreaterThan(x,y)) implies truth)
[Translation] (line 7 of "/home/koepke/TEST/Hales.ftl")
forall v0 ((HeadTerm :: aPackingOfCongruentBallsInEuclideanThreeSpace(v0))
implies truth)
[Translation] (line 10 of "/home/koepke/TEST/Hales.ftl")
forall v0 ((HeadTerm :: v0 = theFaceCenteredCubicPacking) implies
aPackingOfCongruentBallsInEuclideanThreeSpace(v0))
[Translation] (line 16 of "/home/koepke/TEST/Hales.ftl")
aPackingOfCongruentBallsInEuclideanThreeSpace(P)
[Translation] (line 16 of "/home/koepke/TEST/Hales.ftl")
forall v0 ((HeadTerm :: v0 = theDensityOf(P)) implies aRealNumber(v0))
[Translation] (line 18 of "/home/koepke/TEST/Hales.ftl")
forall v0 (aPackingOfCongruentBallsInEuclideanThreeSpace(v0) implies not
isGreaterThan(theDensityOf(v0),theDensityOf(theFaceCenteredCubicPacking)))
[Reasoner] (line 18 of "/home/koepke/TEST/Hales.ftl")
goal:  No packing of congruent balls in Euclidean three space has density
greater than the density of the face centered cubic packing.
[Reasoner] (line 18 of "/home/koepke/TEST/Hales.ftl")
goal failed
[Reasoner] (file "/home/koepke/TEST/Hales.ftl")
verification failed
[Main] (file "/home/koepke/TEST/Hales.ftl")
sections 7 - goals 1 - failed 1 - trivial 0 - proved 0 - equations 0
```

```
[Main] (file "/home/koepke/TEST/Hales.ftl")
symbols 5 - checks 5 - trivial 5 - proved 0 - unfolds 3
[Main] (file "/home/koepke/TEST/Hales.ftl")
parser 00:00.01 - reasoner 00:00.00 - simplifier 00:00.00 - prover 00:00.04/
00:00.00
[Main] (file "/home/koepke/TEST/Hales.ftl")
total 00:00.06
```

Note that the wording of the Theorem in ForTheL is very close to the original. Some differences are:

- whereas the original uses the anaphora "that" to avoid repetition of the word "density", we have to repeat it;
- "face centered" is written without the hyphen "-" since this would be mis-interpreted as a minus symbol.

Further note that `Hales.ftl` is a self-contained ForTheL text. Notions are introduced axiomatically as basically meaningless pattern. ForTheL supports the combination of such patterns into natural language sentences. Internally, statements are translated into first-order logic. The statement of the Theorem becomes:

```
[Translation] (line 18 of "/home/koepke/TEST/Hales.ftl")
forall v0 (aPackingOfCongruentBallsInEuclideanThreeSpace(v0) implies not
isGreaterThan(theDensityOf(v0),theDensityOf(theFaceCenteredCubicPacking)))
```

where `aPackingOfCongruentBallsInEuclideanThreeSpace` is a unary relation symbol, `isGreaterThan` is a binary relation symbol, `theDensityOf` is a unary function symbol, and `theFaceCenteredCubicPacking` is a constant symbol.

The translation is correct within the context of first-order logic where we have the equivalence

$$\neg\exists v_0(P(v_0)\wedge G(v_0))\Leftrightarrow\forall v_0(P(v_0)\rightarrow\neg G(v_0))$$

The controlled natural language ForTheL and its parsing to first-order constructs is the focus of our attention. ForTheL uses many natural language techniques and also some clever tricks and conventions to simulate the reading process of an expert mathematician. ForTheL is a *pattern based language*, where new patterns can be introduced through language extensions (`Signature`) and definitions. Reading consists in identifying patterns and sending them to the corresponding first-order symbols, preserving logical structure.

The intuitive notion of $\geq$ on reals is introduced to the vocabulary of the language by:

```
Signature. x is greater than y is an atom.
```

From this the parser extracts or "learns" a relation pattern like `?,is,greater,than,?` and assigns to it a new binary relation symbol `isGreaterThan` that is put into the vocabulary. Subsequently the parser is regularly looking for this pattern and, if successful, will generate a formula of the form `isGreaterThan(_,_)` with entries which themselves may be results of such pattern recognitions.

Thinking of one's own reading process or that of children indicates, that we are facing an involved trial-and-error-process where the reader is reading a text word by word with perhaps several possible interpretations in mind and with some state of "knowledge" accumulated from earlier parts of the text. We can only expect this tree-like structure of alternative interpretations to collapse to a unique reading when the sentence is completed by a dot "`.`".

Parsing in the Naproche-SAD system can be considered as *Natural Language Processing* (NLP) adjusted to mathematical contexts. So we shall have patterns corresponding to nouns (notions), adjectives and verbs, and also symbolic mathematical patterns like `?,+,?` or `\sqrt{?}` (for $\sqrt{?}$). These patterns will be arranged in grammatical sentence constructs like in nounphrase / verbphrase sentences, or more involved sentences. Note however that nouns, adjectives, etc. are characterized here only by their grammatical function and not by some (English) dictionary. So we can modify Hilberts alleged quote:

> “Man muß jederzeit an Stelle von ’Punkte, Geraden, Ebenen’, ’Tische, Stühle, Bierseidel’ sagen können”

to the extremely formalistic:

> “Man muß jederzeit an Stelle von ’Punkte, Geraden, Ebenen’, ’abcde, $\prec^{\sim}$, a packing of congruent balls in Euclidean three space’ sagen können”.

These are just arbitrary strings like points in a set, and we can just say whether two of these strings are equal or distinct. Although words like congruent, ball, Euclidean space evoke some mathematical meaning, this doesn’t figure at all in the context of `Hales.ftl`.

Luckily one can cover a great deal of mathematics in rather simple grammars, as mathematics describes a “small world” situation of facts about mathematical object in a time-less self-contained environment. This allows to avoid temporal, modal or ambiguous language.

3 Haskell and GHCi: the Module `Core.SourcePos.hs`

Naproche-SAD parses a ForTheL file for its mathematical content. Positions in the file (line/column) are needed, e.g., for user feedback about errors. This is organized via source positions in the input.

Consider the Naproche file `Core/SourcePos.hs`. This file is nearly self-contained and serves us as an entry point into Haskell. We can work with this file in a terminal by

```
$ ghci .../SourcePos.hs
```

Then the various functions defined in the file are available in GHCi.

The file is organized as follows: the file is a definition of the module `SAD.Core.SourcePos`:

```
module SAD.Core.SourcePos
  ( SourcePos (sourceFile, sourceLine, sourceColumn, sourceOffset,
sourceEndOffset),
    SourceRange,
    noPos,
    fileOnlyPos,
    filePos,
    startPos,
    advancePos,
    advancesPos,
    noRangePos,
    rangePos,
    makeRange,
    noRange)
  where
```

The module exports data types like `SourcePos` and `SourceRange` (beginning with CAPITAL letters) and functions like `noPos`, `fileOnlyPos`, ... (beginning with small letters). The definitions of these and other material follow after `where`.

The file then imports material from other files:

```
import qualified Data.List as List
import Isabelle.Library
```

This provides data types and functions to deal with lists and to deal with the interaction with the `Isabelle jedit` proof editor. Note that Haskell has a large set of reserved keywords like `where` or `import` which can be considered as Haskell commands.

3.1 Data types

New data types are introduced like

```
data SourcePos =
  SourcePos {
    sourceFile :: String,
    sourceLine :: Int,
    sourceColumn :: Int,
    sourceOffset :: Int,
    sourceEndOffset :: Int }
  deriving (Eq, Ord)
```

A typical element of this type will be a quintuple like

```
SourcePos "... example.ftl" 35 7 _ _
```

i.e., it denotes a position in the file `example.ftl` like the 35th line and the 7th symbol in the line.

The data type `SourcePos` is built from pre-existing types like `Int` for the integers (also negative) and `String` for strings of symbols. Strings are entered and printed with inverted commas like `"example.ftl"`. Text files can be converted into strings of symbols; the `newline` symbol is represented as `"\n"`. If the file `example.ftl` corresponds to the string

```
Let M denote a set.\nLet f denote a function.\nLet the value of f at x stand
for f[x]. . . .
```

then the word or token `value` begins at the position

```
SourcePos "example.ftl" 3 9 _ _
```

Source positions are used by the parser and proof checker to hint at the locality of errors like in the following message in `Isabelle jedit`:

```
[Parser] (line 40 of "/home/koepke/NAPROCHE/Naproche-SAD/Kelley/Sections_1-
56_original.ftl")
(line 40, column 32):
unexpected z
```

Note that source positions are able to supply the file, line and column coordinates. The file information is important, since a ForTheL text can read in other files.

3.2 Function definitions

The following definitions in the file `SourcePos.hs` define functions for manipulating source positions. E.g., `filePos` is a function that is used to turn a file name `file` into the first source position in that file:

```
filePos :: String -> SourcePos
filePos file = SourcePos file 1 1 1 0
```

The first line defines the type of the function as a function from strings to source positions. The next line define the value of `filePos` at the arbitrary argument `file :: String` by a term in the codomain `SourcePos`. This value is a concrete term in the data type `SourcePos`. Note that `file` is just a variable (= placeholder) used in the definition; we could have used `x` or `y` instead of `file`.

The next group of functions is about advancing positions whilst reading a symbol or a string of symbols:

```
advanceLine line c = if line <= 0 || c /= '\n' then line else line + 1
```

```
advanceColumn column c = if column <= 0 then column else if c == '\n' then 1
else column + 1
...
```

`advanceLine` advances the line number at a new line symbol `\n`. The definition uses a case distinction `if ...then...else` construction. `line` is not changed if the line number is non-positive (meaning no line number) or one is reading a symbol $\neq$ new line. The or disjunction is expressed by `||`. Note that the type of `advanceLine` is not determined as part of the definition, but Haskell derives the type from the form of the definition. One can ask for the type of a function by using the `:type` command in GCHi:

```
*SAD.Core.SourcePos> :type advanceLine
advanceLine :: (Ord a, Num a) => a -> Char -> a
```

The function is *polymorph* in the first argument. It only requires that the type (variable) `a` satisfies `(Ord a, Num a)`. This means that `a` belongs to the type class `Ord` of ordinal types (like the integers) and `a` belongs to the type class `Num` of numerical types where one has the successor function `+1`.

Advancing a source position is obtained by advancing the components:

```
advancePos :: SourcePos -> Char -> SourcePos
advancePos (SourcePos file line column offset endOffset) c =
  SourcePos file
    (advanceLine line c)
    (advanceColumn column c)
    (advanceOffset offset c)
    endOffset
```

Note that this definition uses a pattern matching: the first argument to have the form `SourcePos file line column offset endOffset` which instantiates variables for the `file` string and for the other numbers. Note also that we have to use brackets around the pattern since otherwise `advancePos` would try to take `SourcePos` and `file` as its arguments.

The next function definition involves the function `foldl` ("fold left"):

```
advancesPos :: SourcePos -> String -> SourcePos
advancesPos (SourcePos file line column offset endOffset) s =
  SourcePos file
    (foldl advanceLine line s)
    (foldl advanceColumn column s)
    (foldl advanceOffset offset s)
    endOffset
```

`foldl x y z` expresses the iteration of the binary function `x` along the list `z` starting from the initial value `y`. Such operations are more familiar from arithmetic: the finite sume $\sum$ of a list of arguments is obtained by iterating the $+$-operation. The sum of 1 to 10 can be computed in GHCi by

```
*SAD.Core.SourcePos> foldl (+) 0 [1,2,3,4,5,6,7,8,9,10]
55
```

This corresponds to bracketing in the evaluation. The effect is more obvious when iterating the $-$-operation: Consider the term $-1-2$:

```
*SAD.Core.SourcePos> foldl (-) 0 [1,2]
-3
*SAD.Core.SourcePos> foldr (-) 0 [1,2]
-1
```

Left folding corresponds to left-bracketing $0-1-2=(0-1)-2=-3$; right folding corresponds to right bracketing $1-2-0=1-(2-0)=-1$. We check the type of `foldl`:

```
*SAD.Core.SourcePos> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

Lists `[ . . . ]` are foldable. Note that `t` is a *type transformer* which makes the type `t a` out of the type `a`. The type former `[ . . . ]` forms lists of a given type. We could also define `foldl` for lists by recursion on that list:

```
*SAD.Core.SourcePos> let {fl op start [] = start; fl op start (a:as) = op (fl
op start as) a}
*SAD.Core.SourcePos> fl (+) 0 [1,2,3,4,5,6,7,8,9,10]
55
```

Note how we make a multiline definition using a `let{ . . . ; . . . }` multiline construct. Comparing the types we see that `fl` is a specialization of `foldl`:

```
*SAD.Core.SourcePos> :t fl
fl :: (t1 -> t -> t1) -> t1 -> [t] -> t1
*SAD.Core.SourcePos> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
*SAD.Core.SourcePos> fl (-) 0 [1,2,3,4,5,6,7,8,9,10]
-55
*SAD.Core.SourcePos> foldl (-) 0 [1,2,3,4,5,6,7,8,9,10]
-55
```

Note that `foldl` is a *higher order function* since it takes functions as arguments. Haskell programming is based on using lots of higher order functions. Some of them like `foldl` correspond to higher order notions and operations in mathematics. In this case the somewhat unsharp correspondance would be "iteration of a binary operation". Whereas in mathematics a general idea of iteration may carry sufficient information, a computer language must be specific about details as expressed in the type property

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

3.3 Printing values

GHCi tries to print out function values. Values of a type `type` can only be printed if there is a `show` function for that type. Also such types are registered to be instances of the type class `Show`.

```
instance Show SourcePos where
  show (SourcePos file line column _ _) =
    if null showName then showDetails
    else if null showDetails then showName
    else showName ++ " " ++ showDetails
    where
      detail a i = if i <= 0 then "" else a ++ " " ++ show i
      details = [detail "line" line, detail "column" column]
      showDetails =
        case filter (not . null) details of
          [] -> ""
          ds -> "(" ++ commas ds ++ ")"
      showName = if null file then "" else "\"" ++ file ++ "\""
```

The result of a show operation is a string which is then printed out:

```
*SAD.Core.SourcePos> :t show
show :: Show a => a -> String
```

So `show (SourcePos file line column _ _)` builds a string consisting of the components of the source position that one wants to see. `file` is a string that can be used in string operations like `++`; to show an integer one relies on the predefined show operation for the type `Int`. One could change the show operation for the type `SourcePos` by modifying the above definition. At the moment, the show function mainly shows the line and column information:

```
*SAD.Core.SourcePos> startPos
(line 1, column 1)
*SAD.Core.SourcePos> advancesPos startPos "hallo"
(line 1, column 6)
*SAD.Core.SourcePos> advancesPos startPos "hal\nlo"
(line 2, column 3)
```

A show function for a type could also be generated automatically by putting `deriving Show` at the end of a type. In `SourcePos.hs` we only registered `SourcePos` in the classes `Eq` and `Ord`. We now use the automatic derivation:

```
data SourcePos =
  SourcePos {
     . . . }
  deriving (Eq, Ord, Show)
```

This leads to an error of two instances of `show` defined for one type:

```
*SAD.Core.SourcePos> :reload
[1 of 1] Compiling SAD.Core.SourcePos ( SourcePos.hs, interpreted )

SourcePos.hs:35:22: error:
    Duplicate instance declarations:
      instance Show SourcePos -- Defined at SourcePos.hs:35:22
      instance Show SourcePos -- Defined at SourcePos.hs:94:10
Failed, modules loaded: none.
```

After commenting out the definition at `94:10` using `{- ... -}` we get the automatic show function:

```
Prelude> :reload
[1 of 1] Compiling SAD.Core.SourcePos ( SourcePos.hs, interpreted )
Ok, modules loaded: SAD.Core.SourcePos.
*SAD.Core.SourcePos> startPos
SourcePos {sourceFile = "", sourceLine = 1, sourceColumn = 1, sourceOffset = 1,
sourceEndOffset = 0}
*SAD.Core.SourcePos> advancesPos startPos "hallo"
SourcePos {sourceFile = "", sourceLine = 1, sourceColumn = 6, sourceOffset = 6,
sourceEndOffset = 0}
*SAD.Core.SourcePos> advancesPos startPos "hal\nlo"
SourcePos {sourceFile = "", sourceLine = 2, sourceColumn = 3, sourceOffset = 7,
sourceEndOffset = 0}
```

4 Tokenizing and `Parser.Token.hs`

The ForThel input file to the system Naproche-SAD is first read in as a (long) string (= list of symbols). Then it is translated or tokenized into a sequence of tokens, by eliminating whitespace and packaging comments as one token. Regular tokens are longest possible sequences of *lexems* defined by

```
isLexem :: Char -> Bool
```

```
isLexem c = isAscii c && isAlphaNum c || c == '_'
```

Note that this means

```
(isAscii c && isAlphaNum c)   ||   (c == '_')
```

by the priorities of the logical `&&` ("and") and `||` ("or") in Haskell. Other symbolic material is cut up into single symbols.

We open the file `.../Parser/Token.hs` in GHCi. In the context of Naproche-SAD `tokenize` will be used in the form `tokenize (filePos file) input`. Let us first give some examples of tokenizing:

```
koepke@dell:~/NAPROCHE/Naproche-SAD/src/SAD/Parser$ ghci Token.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling SAD.Parser.Token ( Token.hs, interpreted )
Ok, modules loaded: SAD.Parser.Token.
*SAD.Parser.Token> tokenize (filePos "") "The equation."

[The(line 1, column 1),equation(line 1, column 5),.(line 1, column 13),]
*SAD.Parser.Token>
*SAD.Parser.Token> tokenize (filePos "FILENAME") "The equation."
[The"FILENAME" (line 1, column 1),equation"FILENAME" (line 1, column 5),
."FILENAME" (line 1, column 13),]
*SAD.Parser.Token> tokenize (filePos "") "The equation a = b."
[The(line 1, column 1),equation(line 1, column 5),a(line 1, column 14),=(line
1, column 16),b(line 1, column 18),.(line 1, column 19),]
*SAD.Parser.Token> tokenize (filePos "") "The equation a != b."
[The(line 1, column 1),equation(line 1, column 5),a(line 1, column 14),!(line
1, column 16),=(line 1, column 17),b(line 1, column 19),.(line 1, column 20),]
```

If one wants to ignore the coordinates of tokens, one can tokenize with `noPos` which was defined as

```
noPos :: SourcePos
noPos = SourcePos "" 0 0 0 0
```

We get

```
*SAD.Parser.Token> tokenize noPos "The equation a != b."
[The,equation,a,!,=,b,.,]
```

Naproche-SAD handles enormous amounts of data so that the human user needs to filter out the information to be focussed on. Note, however, that the program might heavily use the implicit data which is not displayed by `show` commands.

4.1 Symbols and lists

The module `Token.hs` imports powerful functions for characters and lists from the modules `Data.Char` and `Data.List`.

```
import Data.Char
import Data.List
```

(The second import appears to be redundant) The data type

```
data Token =
  Token {
    tokenText :: String,
    tokenPos :: SourcePos,
```

```
      tokenWhiteSpace :: Bool,
      tokenProper :: Bool} |
    EOF {tokenPos :: SourcePos}
```

has a Boolean flag `tokenWhiteSpace` indicating that there has been whitespace directly before the token. Such flags can be handled by extra arguments or by some state bit. Probably this flag will be used to see whether symbols are separated by whitespace or directly adjacent.

The function `tokenize` yields the accumulated result of an auxiliary function `posToken`; `posToken` recursively scans through the input string, using pattern matching cases. `posToken` has the additional Boolean argument to deal with the whitespace arguments of tokens.

4.2 Tokenization

To explain the definition of `tokenize` and `posToken` we have inserted comments into the code:

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
-- the second argument is the status bit for "whitespace before"
  where
-- look for a non-empty string of lexems;
-- then make this string into a proper token ("True") with the same
-- whitespace flag; the rest of the input is tokenized with the
-- flag set to False
    posToken pos ws s
      | not (null lexem) =
          makeToken lexem pos ws True : posToken (advancesPos pos lexem) False
rest
      where (lexem, rest) = span isLexem s
-- look for a non-empty string of blanks; then forget this whitespace and
-- continue
    posToken pos _ s
      | not (null white) = posToken (advancesPos pos white) True rest
      where (white, rest) = span isSpace s
-- look for a comment marker (#) and turn the comment into an improper token
    posToken pos ws s@('#':_) =
      makeToken comment pos False False : posToken (advancesPos pos comment) ws
rest
      where (comment, rest) = break (== '\n') s
-- if the above cases fail and the input is non-trivial then it must begin
-- with a symbol which is turned into a token
    posToken pos ws (c:cs) =
      makeToken [c] pos ws True : posToken (advancePos pos c) False cs
-- otherwise the input is empty and an EOF token is generated
    posToken pos _ _ = [EOF pos]
```

Note that the tokenization could be modified easily. E.g., if we want to tokenize adequately for LaTeX we should not just look for longest strings of lexems, but for longest strings starting with \, followed by lexems.

5 Reading files

The input strings for tokenization and further processing should come from input files. The `Main.hs` file initializes the I/O:

```
main :: IO ()
main  = do
```

```
    -- setup stdin/stdout
    File.setup IO.stdin
    File.setup IO.stdout
    File.setup IO.stderr
    IO.hSetBuffering IO.stdout IO.LineBuffering
    IO.hSetBuffering IO.stderr IO.LineBuffering
   ...
```

If the (many) options are fine and no errors occur, `main` calls `mainBody`:

```
  mainBody oldTextRef (opts0, text0))
```

Similarly `mainBody` may call `readText`:

```
  mainBody :: IORef Text -> ([Instr], [Text]) -> IO ()
  mainBody oldTextRef (opts0, text0) = do
    startTime <- getCurrentTime

    oldText <- readIORef oldTextRef
    -- parse input text
    text1 <- fmap TextRoot $ readText (Instr.askString Instr.Library "." opts0)
  text0
   ...
```

`readText` is defined in `Import/Reader.hs`:

```
  readText :: String -> [Text] -> IO [Text]
  readText pathToLibrary text0 = do
    pide <- Message.pideContext
    (text, reports) <- reader pathToLibrary [] [State (initFS pide) noTokens
  noPos] text0
    when (isJust pide) $ Message.reports reports
    return text
```

This relies on `reader` which is defined in the same module. If all goes well, `reader` does read the intended input file:

```
  reader pathToLibrary doneFiles (pState:states) [TextInstr _ (Instr.String
  Instr.File file)] = do
    text <-
      catch (if null file then getContents else File.read file)
        (Message.errorParser (fileOnlyPos file) . ioeGetErrorString)
    (newText, newState) <- reader0 (filePos file) text pState
    reader pathToLibrary (file:doneFiles) (newState:pState:states) newText
```

basically by `doing`, with some provisos and error checking:

```
  text <- File.read file
```

where `file` is a path to the input file. This `text` is fed into `reader0` which calls the tokenization and its parsing by the `forthel` parser:

```
  reader0 :: SourcePos -> String -> State FState -> IO ([Text], State FState)
  reader0 pos text pState = do
    let tokens0 = tokenize pos text
    Message.reports $ concatMap tokenReports tokens0
    let tokens = filter properToken tokens0
        st = State ((stUser pState) { tvrExpr = [] }) tokens noPos
```

```
launchParser forthel st
```

5.1 I/O in GHCi

In GHCi I/O is already initialized, and we can get the above mechanisms rather easily. We are interested in the file `powerset.ftl` in the home directory which proves that a powerset is strictly bigger than the original set.

```
[synonym subset/-s] [synonym surject/-s]

Let M denote a set.
Let f denote a function.
Let the value of f at x stand for f[x].
Let f is defined on M stand for Dom(f) = M.
Let the domain of f stand for Dom(f).

Axiom. The value of f at any element of the domain of f is a set.

Definition.
A subset of M is a set N such that every element of N is an element of M.

Definition.
The powerset of M is the set of subsets of M.

Definition.
f surjects onto M iff every element of M is equal to the value of f at some
element of the domain of f.

Proposition.
No function that is defined on M surjects onto the powerset of M.
Proof.
Assume the contrary. Take a function f that is defined on M and surjects onto
the powerset of M.
Define N = { x in M | x is not an element of f[x] }.
Then N is not equal to the value of f at any element of M.
Contradiction. qed.
```

In GHCi we can quickly read this file into a string variable and print it:

```
*SAD.Import.Reader> text <- File.read "/home/koepke/powerset.ftl"
*SAD.Import.Reader> text
"[synonym subset/-s] [synonym surject/-s]\n\nLet M denote a set.\nLet
f denote a function.\nLet the value of f at x stand for f[x].\nLet f
is defined on M stand for Dom(f) = M.\nLet the domain of f stand for
Dom(f).\n\nAxiom. The value of f at any element of the domain of f is a
set.\n\nDefinition.\nA subset of M is a set N such that every element of N is
an element of M.\n\nDefinition.\nThe powerset of M is the set of subsets of
M.\n\nDefinition.\nf surjects onto M iff every element of M is equal to the
value of f at some element of the domain of f.\n\nProposition.\nNo function
that is defined on M surjects onto the powerset of M.\nProof.\nAssume the
contrary. Take a function f that is defined on M and surjects onto the powerset
of M.\nDefine N = { x in M | x is not an element of f[x] }.\nThen N is not
equal to the value of f at any element of M.\nContradiction. qed.\n"
```

We can moreover tokenize the text:

```
*SAD.Import.Reader> tokenize noPos text
```

```
[[,synonym,subset,/,-,s,],[,synonym,surject,/,-,s,],Let,M,denote,a,set,.,
Let,f,denote,a,function,.,Let,the,value,of,f,at,x,stand,for,f,[,x,],.,Let,f,
is,defined,on,M,stand,for,Dom,(,f,),=,M,.,Let,the,domain,of,f,stand,for,Dom,
(,f,),.,Axiom,.,The,value,of,f,at,any,element,of,the,domain,of,f,is,a,set,
.,Definition,.,A,subset,of,M,is,a,set,N,such,that,every,element,of,N,is,an,
element,of,M,.,Definition,.,The,powerset,of,M,is,the,set,of,subsets,of,M,.,
Definition,.,f,surjects,onto,M,iff,every,element,of,M,is,equal,to,the,value,
of,f,at,some,element,of,the,domain,of,f,.,Proposition,.,No,function,that,is,
defined,on,M,surjects,onto,the,powerset,of,M,.,Proof,.,Assume,the,contrary,.,
Take,a,function,f,that,is,defined,on,M,and,surjects,onto,the,powerset,of,M,.,
Define,N,=,{,x,in,M,|,x,is,not,an,element,of,f,[,x,],},.,Then,N,is,not,equal,
to,the,value,of,f,at,any,element,of,M,.,Contradiction,.,qed,.,]
```

Before the tokenization is fed into parsing one can filter out the proper tokens using the *higher order function* `filter`:

```
*SAD.Import.Reader> :t filter
filter :: (a -> Bool) -> [a] -> [a]
*SAD.Import.Reader> :t properToken
properToken :: Token -> Bool
```

`properToken` was defined in the module `Token.hs` from the data type `Token`:

```
properToken :: Token -> Bool
properToken Token {tokenProper} = tokenProper
properToken EOF {} = True
```

With that:

```
*SAD.Import.Reader> let tokens = tokenize noPos text
*SAD.Import.Reader> filter properToken tokens
[[,synonym,subset,/,-,s,],[,synonym,surject,/,-,s,],Let,M,denote,a,set,.,
Let,f,denote,a,function,.,Let,the,value,of,f,at,x,stand,for,f,[,x,],.,Let,f,
is,defined,on,M,stand,for,Dom,(,f,),=,M,.,Let,the,domain,of,f,stand,for,Dom,
(,f,),.,Axiom,.,The,value,of,f,at,any,element,of,the,domain,of,f,is,a,set,
.,Definition,.,A,subset,of,M,is,a,set,N,such,that,every,element,of,N,is,an,
element,of,M,.,Definition,.,The,powerset,of,M,is,the,set,of,subsets,of,M,.,
Definition,.,f,surjects,onto,M,iff,every,element,of,M,is,equal,to,the,value,
of,f,at,some,element,of,the,domain,of,f,.,Proposition,.,No,function,that,is,
defined,on,M,surjects,onto,the,powerset,of,M,.,Proof,.,Assume,the,contrary,.,
Take,a,function,f,that,is,defined,on,M,and,surjects,onto,the,powerset,of,M,.,
Define,N,=,{,x,in,M,|,x,is,not,an,element,of,f,[,x,],},.,Then,N,is,not,equal,
to,the,value,of,f,at,any,element,of,M,.,Contradiction,.,qed,.,]
```

Apparently, all tokens were proper tokens (and not comments, e.g.).

6 Parsing

After tokenization the sequence of tokens has to be read or parsed into the system. The result of parsing should be in some expected type or an error. Parsing is composed of many subparsings. Parsing an (expected) text requires parsing of sentences which in term requires parsing certain parts of sentences etc.

6.1 Parsers

Let us quote from the well-known paper *Monadic Parsing* by G. Hutton and E. Meijer:

We begin by defining a type for parsers:

```
newtype Parser a = Parser (String -> [(a,String)])
```

That is, a parser is a function that takes a string of characters as its argument, and returns a list of results. The convention is that the empty list of results denotes failure of a parser, and that non-empty lists denote success. In the case of success, each result is a pair whose first component is a value of type a produced by parsing and processing a prefix of the argument string, and whose second component is the unparsed suffix of the argument string. Returning a list of results allows us to build parsers for ambiguous grammars, with many results being returned if the argument string can be parsed in many different ways.

In Naproche-SAD parsers are defined similarly but with considerably more technical complications:

- parsers are reading token lists instead of strings, including position information;
- parsers modify themselves by reading a text which contains definitions and assumptions; this is modeled by modifying some internal state of the parser, hence parsers will work on their own state;
- the state of parsing is not only defined by the yet unread tokens, but by some value (state) in some specific datatype that contains data accumulated along the reading;
- there is a lot of error handling, since their may be errors in the input and parsers may (intentionally) fail.

The Naproche-SAD definition of `Parser` proceeds as follows. Instead of the simple parsing states above, given by the string of unparsed symbols, the state now has two components: `stInput` is a list of unparsed symbols, and `stUser` is a generic state which could be used to accumulate information along the parsing process.

```
-- Parser state
data State st = State {
  stUser  :: st,
  stInput :: [Token],
  lastPosition :: SourcePos}
```

As above, a parsing instance yields a pair of a result in some type `a` together with a `State`:

```
data ParseResult st a = PR { prResult :: a, prState :: State st }
```

A parser, fed with a certain token list, should now produce a member of

```
[ParseResult st a]
```

In a complicated parsing network such lists will be given to another parser etc. We make use of a particular device of *continuation passing* where a new functional argument is added to functions. When we have a unary function $f(x)=y$ we can transform this into a binary function

$$f'(x,c)=c(y)=c(f(x))$$

where c is called a continuation and stands for arbitrary further operations on the value. Obviously the original function can be recovered from the binary one since $f(x)=f'(x,\mathrm{id})$. This simple idea has several variants. One could, e.g., in case the function value is a tuple with several components, use a tuple of continuation functions. Setting up the continuations cleverly will help with operations of functions like the composition of parsers.

Parsing in Naproche-SAD makes use of three continuations which also deal with errors:

```
type Continuation st a b =
  ParseError -> [ParseResult st a] -> [ParseResult st a] -> b
type EmptyFail    b = ParseError -> b
```

```
type ConsumedFail b = ParseError -> b
```

Parsers then are defined to have a general shape as:

```
-- Continutation passing style ambiguity parser
newtype Parser st a = Parser {runParser :: forall b .
     State st
  -> Continuation st a b
  -> ConsumedFail b
  -> EmptyFail b
  -> b }
```

Let us look at a simple parser that consumes one token and applies a function to it:

```
---- parse the current token
tokenPrim :: (Token -> Maybe a) -> Parser st a
tokenPrim test = Parser $ \(State st input _) ok _ eerr ->
  case uncons input of
    Nothing     -> eerr $ unexpectError "" noPos
    Just (t,ts) -> case guard (not $ isEOF t) >> test t of
      Just x  ->
        let newstate = State st ts (tokenPos t)
            newerr   = newErrorUnknown $ tokenPos t
        in  seq newstate $ ok newerr [] . pure $ PR x newstate
      Nothing -> eerr $ unexpectError (showToken t) (tokenPos t)
```

The input list of tokens is given by `input`. Normally, the input is "unconsed" as `input = t : ts`. The token function `test` is applied to the token `t` and yields a value `x`. The new `State` is given by the tale `ts` and the parsing result is

```
PR x $ State st ts (tokenPos t)
```

This straightforward process is surrounded by failure management using the `Maybe` monad, several error functions, and mapping the parsing result by the continuation function `ok`. Note that the "User state" `st` has not been touched by this parser. It will only be used, once a convenient data type has been defined.

Parsers with their continuation functions satisfy important algebraic laws of being *monads* and the like. The internals of those operations for our kind of parsers look horrible and are best ignored for our purposes:

```
tryParses :: (a -> Parser st b) -> Continuation st b c -> ConsumedFail c
          -> EmptyFail c -> ParseError -> [ParseResult st a]
          -> [ParseResult st a] ->  c
tryParses f ok cerr eerr err eok cok = accumE err [] [] [] [] eok
  where
    accumE acc_err acc_eok acc_cok acc_cerr acc_eerr ((PR a st'):rs) =
      let fok ferr feok fcok =
            accumE (acc_err <++> ferr) (reverse feok ++ acc_eok)
                   (reverse fcok ++ acc_cok) acc_cerr acc_eerr rs
          fcerr err' =
            accumE acc_err acc_eok acc_cok (err':acc_cerr) acc_eerr rs
          feerr err' =
            accumE acc_err acc_eok acc_cok acc_cerr (err':acc_eerr) rs
      in  runParser (f a) st' fok fcerr feerr
    accumE acc_err acc_eok acc_cok acc_cerr acc_eerr [] =
      accumC acc_err acc_eok acc_cok acc_cerr acc_eerr cok
```

```
    accumC acc_err acc_eok acc_cok acc_cerr acc_eerr ((PR a st'):rs) =
      let fok ferr feok fcok =
            accumC (acc_err <+> ferr) acc_eok
              (reverse feok ++ reverse fcok ++ acc_cok) acc_cerr acc_eerr rs
          fcerr err' =
            accumC acc_err acc_eok acc_cok (err':acc_cerr) acc_eerr rs
          feerr err' =
            accumC acc_err acc_eok acc_cok (err':acc_cerr) acc_eerr rs
      in  runParser (f a) st' fok fcerr feerr
    accumC acc_err acc_eok acc_cok acc_cerr acc_eerr []
      | (not $ null acc_eok)  || (not $ null acc_cok)  = ok acc_err (reverse
acc_eok) (reverse acc_cok)
      | (not $ null acc_eerr) = eerr $ foldl' (<++>) err $ acc_eerr ++ acc_cerr
      | (not $ null acc_cerr) = cerr $ foldl' (<++>) err $ acc_cerr
      | otherwise = error "tryParses: parser has empty result"
```

6.2 Running parsers

In Naproche-SAD parsers work embedded in the system and are difficult to experiment with. We use auxiliary devices defined in `Base.hs`. We want to apply a parser to a `State` which for simplicity is a list of tokens that we want to parse. The output should be a list of parse results or a `ParseError`.

```
data Reply a = Ok [a] | Error ParseError
```

The parsing is invoked by

```
---- running the parser
runP :: Parser st a -> State st -> Reply (ParseResult st a)
runP p st = runParser p st ok cerr eerr
  where
    ok _ eok cok = Ok $ eok ++ cok
    cerr err     = Error err
    eerr err     = Error err
```

The local functions `ok`, `cerr` and `eerr` correspond to the identity function id in our simplicistic continuation example above.

6.3 Test tools in `Test.ParserTest`

We use `runP` within an ad hoc module `../Test/ParserTest.hs` for parser testing. It can be extended to allow file I/O and show functions for more sophisticated parsers. We use ad hoc functions `showSt`, `showPr`, and `showRp` to display elements of `State`, `ParseResult` and `Reply`.

```
{-
Authors: Peter Koepke (2019)

ParserTest - Test tools for parsers

-}

module SAD.Test.ParserTest where

import SAD.Core.SourcePos
import SAD.Parser.Base
import SAD.Parser.Token
```

```
import SAD.Parser.Error
import SAD.Parser.Primitives

import Data.Char
import Data.List

import Control.Monad
import Data.Maybe (isJust, fromJust)
import Debug.Trace

showSt (State stat tokens position) = show tokens
showPr (PR result stat) = (show result) ++ " " ++ (showSt stat)
showRp (Ok list) = map showPr list
-- showRp (Error e) = show e
```

The name of the module defined inside the file has to match the name of the file. The file can be run in GHCi. It also import `runP` and yields partial results like:

```
*SAD.Parser.Primitives> showRp $ runP anyToken (State 5 (tokenize noPos "hallo
world") noPos)
["\"hallo\"   [world,]"]
```

The parser `anyToken` is taking any token off the tokenlist without further action. We call also play with parser combinators like running `anyToken` two times in a row which is obtained by the monadic `>>`:

```
*SAD.Test.ParserTest> showRp $ runP (anyToken >> anyToken) (State 5 (tokenize
noPos "hallo world") noPos)
["\"world\" []"]
```

Note that the general framework for parsers and parsing is given by the basic definitions in `Parser/Base.hs`. We can take them as a black box, so that we do not have to worry about the exact and difficult details of program flow. We can instead use, combine and modify existing parsers outside the box. We expect that this approach will allow to probe and modify ForTheL parsing for many purposes.

7 Basic parsers

We want to test the parser

```
tokenPrim :: (Token -> Maybe a) -> Parser st a
```

that we have already encountered above. This requires a function of type `Token -> Maybe a`. Let us extract the first symbol of a token. Recall:

```
data Token =
  Token {
    tokenText :: String,
    tokenPos :: SourcePos,
    tokenWhiteSpace :: Bool,
    tokenProper :: Bool} |
  EOF {tokenPos :: SourcePos}
```

The field names like `tokenText` also serve as projections onto the corresponding coordinate.

```
*SAD.Test.ParserTest> :t tokenText
```

```
tokenText :: Token -> String
```

Then the function composition `head . tokenText`, using the infix composition operator `(.)`, looks right, except that the value would be in `Char`. The data type for `Maybe a` is:

```
*SAD.Test.ParserTest> :info Maybe
data Maybe a = Nothing | Just a         -- Defined in ‘GHC.Base’
```

So we compose with the constructor `Just` and have got the right type:

```
*SAD.Test.ParserTest> :t Just . head . tokenText
Just . head . tokenText :: Token -> Maybe Char
```

Now `tokenPrim (Just . head . tokenText)` is the desired parser:

```
*SAD.Test.ParserTest> :t tokenPrim (Just . head . tokenText)
tokenPrim (Just . head . tokenText) :: Parser st Char
*SAD.Test.ParserTest> :t tokenPrim $ Just . head . tokenText
tokenPrim $ Just . head . tokenText :: Parser st Char
```

This also demonstrates that we can use `$` for a bracket which extends to the end of the input. Our new parser indeed extracts the first character of the first token:

```
*SAD.Test.ParserTest> showRp $ runP (tokenPrim $ Just . head . tokenText)
(State 5 (tokenize noPos "high world") noPos)
["'h' [world,]"]
```

Substituting `length` for `head` we similarly get at the length of the first token:

```
*SAD.Test.ParserTest> showRp $ runP (tokenPrim $ Just . length . tokenText)
(State 5 (tokenize noPos "high world") noPos)
["4 [world,]"]
```

Let us now go through other parsers introduced in `Parser.Primitives.hs`. The parser `eof` checks for an end-of-file token and is built very similar to `tokenPrim`.

```
---- parse end of input
eof :: Parser st ()
eof = Parser $ \(State st input _) ok _ eerr ->
  case uncons input of
    Nothing -> eerr $ unexpectError "" noPos
    Just (t, ts) ->
      if isEOF t
      then
        let newstate = State st ts (tokenPos t)
            newerr   = newErrorUnknown $ tokenPos t
        in  seq newstate $ ok newerr [] . pure $ PR () newstate
      else eerr $ unexpectError (showToken t) (tokenPos t)
```

The first example throws an error, the second succeeds:

```
*SAD.Test.ParserTest> showRp $ runP eof (State 5 (tokenize noPos "Naproche
SAD") noPos)
":\nunexpected Naproche"
*SAD.Test.ParserTest> showRp $ runP eof (State 5 (tokenize noPos "") noPos)
"[\"() []\"]"
```

The next parser `satisfy` is an instance of `tokenPrim` where the first token is subjected to the test function:

```
*SAD.Test.ParserTest> let prTest pr tk = let s = showToken tk in guard (pr s)
>> return s
```

Note that the predicate `pr` is added as an argument since it is not supplied by the outer function definition of `satisfy`. Let us look at the function types.

```
*SAD.Test.ParserTest> :t prTest
prTest
  :: (Monad m, GHC.Base.Alternative m) =>
     (String -> Bool) -> Token -> m String
```

Given a test function `:: String -> Bool` this function has type `Token -> m String`, where `m` is a variable for some monad. In the context of the definition of `satisfy` `m` will be unified with the `Maybe` monad. It the string `s = showToken tk` satisfies the predicate `pr` then the guard function will succeed and pass control to `return s` which will output that string. Otherwise `guard` throws an error that will be handled by the further error processing:

```
*SAD.Test.ParserTest> :t guard
guard :: GHC.Base.Alternative f => Bool -> f ()
*SAD.Test.ParserTest> :t guard True
guard True :: GHC.Base.Alternative f => f ()
*SAD.Test.ParserTest> :t guard False
guard False :: GHC.Base.Alternative f => f ()
*SAD.Test.ParserTest> guard True
*SAD.Test.ParserTest> guard False
*** Exception: user error (mzero)
*SAD.Test.ParserTest> return "hallo"
"hallo"
*SAD.Test.ParserTest> guard True >> return "hallo"
"hallo"
*SAD.Test.ParserTest> guard False >> return "hallo"
*** Exception: user error (mzero)
```

Here is the full code for `satisfy`:

```
satisfy :: (String -> Bool) -> Parser st String
satisfy pr = tokenPrim prTest
  where
    prTest tk = let s = showToken tk in guard (pr s) >> return s
```

We can see `satisfy` in action in the next parser `word`

```
---- check if the current token is a word
word :: Parser st String
word = satisfy $ \tk -> all isAlpha tk


*SAD.Test.ParserTest> showRp $ runP word (State 5 (tokenize noPos "Naproche
SAD") noPos)
"[\"\\\"Naproche\\\" [SAD,]\"]"
*SAD.Test.ParserTest> showRp $ runP word (State 5 (tokenize noPos "1234 SAD")
noPos)
":\nunexpected 1234"
```

We can also check for specific words. Since natural language text use capitalization, whereas internally we work with lower cases, `wdToken` is defined as:

```
---- check if the current token is equal to s after mapping to lowercase
```

```
{-# INLINE wdToken #-}
wdToken :: String -> Parser st ()
wdToken s = void $ satisfy $ \tk -> s == map toLower tk

*SAD.Test.ParserTest> showRp $ runP (wdToken "forthel") (State 5 (tokenize
noPos "ForTheL SAD") noPos)
"[\"() [SAD,]\"]"
*SAD.Test.ParserTest> showRp $ runP (wdToken "forthel") (State 5 (tokenize
noPos "1234 SAD") noPos)
":\nunexpected 1234"
```

We can also check whether the string of the token is an element of a list of strings:

```
---- check if the current token is equal to some element of ls after
---- mapping to lowercase
{-# INLINE wdTokenOf #-}
wdTokenOf :: [String] -> Parser st ()
wdTokenOf ls = void $ satisfy $ \tk -> map toLower tk `elem` ls
```

8 Some Parser combinators

The module `Parser.Base` registered our type `Parser` as instances of the type classes `Functor`, `Applicative`, `Monad`, `Alternative` and `MonadPlus`. These type classes all possess certain operators on their elements which satisfy some natural mostly algebraic properties. The class `monad` requires an operation `return` and is defined in this case as:

```
instance Monad (Parser st) where
  return x = Parser $ \st ok _ _ ->
      ok (newErrorUnknown (stPosition st)) [PR x st] []
```

Within the context of our parsers we are basically returning the argument `x` as a value. For reasons of formating and since we are working with continuation passing, the `x` is packaged within a cloud of canonical non-informative components.

A characteristic operation for monads is the binary operation `>>=` where `p >>= f` is pronounced `p` *binds* `f` (???). The type of `bind` is:

```
*SAD.Test.ParserTest> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Let us consider an application in the context of simple parsing where the type `a = String`. The parser `word` outputs the first token as a string. Now we can insert that string into the parser `wdToken` and check whether the string of the second token is the same. This means that the combination `word >>= wdtoken` should check, whether a text starts with two equal words (all written in small letters).

```
*SAD.Test.ParserTest> :t word
word :: Parser st String
*SAD.Test.ParserTest> :t wdToken
wdToken :: String -> Parser st ()
*SAD.Test.ParserTest> :t word >>= wdToken
word >>= wdToken :: Parser st ()
*SAD.Test.ParserTest> showRp $ runP (word >>= wdToken) (State 5 (tokenize noPos
"naproche naproche") noPos)
"[\"() []\"]"
*SAD.Test.ParserTest> showRp $ runP (word >>= wdToken) (State 5 (tokenize noPos
"naproche sad") noPos)
```

```
":\nunexpected sad"
```

This demonstrates a typical way to combine parsing where later parsing often depends on the result of previous parsing.

A monad assumes that `return` and `>>=` satisfy the following identities:

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> (k x >>= h))  =  (m >>= k) >>= h
```

`return` acts like a neutral element with respect to `>>=`; the third law is a kind of associativity. Such laws are (apparently) also used by the Haskell compiler to optimize code.

If there is no value-argument binding between the first and the second parser, one can use the simpler binder `>>` which only means: apply the first parser and then the second. `>>` can be defined from `>>=`:

```
m >> k = m >>= \_ -> k
```

where `\_ -> k` is the constant lamda term with value `k` for arbitrary arguments. We can use `>>` to check for a certain sequence of words at the beginning of a text:

```
*SAD.Test.ParserTest> showRp $ runP (wdToken "one" >> wdToken "two" >> wdToken
"three") (State 5 (tokenize noPos "one two three four") noPos)

"[\"() [four,]\"]"
```

Note the empty output `()` (pronounced *unit*) signals the success of the parsing.

Besides elegant possibilities to combine *monadic* parsers, we shall also use monads to carry out temporal modifications of states: `Parser st` handles a state of some data type `st`.

A `MonadPlus` has a specific operation `mplus` that can be applied to parsers.

```
*SAD.Test.ParserTest> :t wdToken "one" `mplus` wdToken "two"
wdToken "one" `mplus` wdToken "two" :: Parser st ()
```

As a parser combinator this operation is usually written as follows:

```
------  Choose in LL1 fashion
infixr 2 <|>
{-# INLINE (<|>) #-}
(<|>) :: Parser st a -> Parser st a -> Parser st a
(<|>) = mplus
```

First experiments yield:

```
*SAD.Test.ParserTest> :t wdToken "one" <|> wdToken "two"
wdToken "one" <|> wdToken "two" :: Parser st ()
*SAD.Test.ParserTest> showRp $ runP (wdToken "one" <|>  wdToken "two") (State 5
(tokenize noPos "one two three four") noPos)
"[\"() [two,three,four,]\"]"
*SAD.Test.ParserTest> showRp $ runP (wdToken "one" <|>  wdToken "two") (State 5
(tokenize noPos "two one three four") noPos)
"[\"() [one,three,four,]\"]"
*SAD.Test.ParserTest> showRp $ runP (wdToken "one" <|>  wdToken "two") (State 5
(tokenize noPos "three one two four") noPos)
":\nunexpected three"
```

The `<|>` combinator looks at the next **1** Token and then chooses preferably the **left** alternative of the two parsers. This style of parsing or grammar is classified as LL1, for “left, 1 look ahead”. This is illustrated by:

```
*SAD.Test.ParserTest> showRp $ runP ((wdToken "one" >> wdToken "one") <|>
(wdToken "one" >> wdToken "two"))  (State 5 (tokenize noPos "one two three
four") noPos)
":\nunexpected two"
*SAD.Test.ParserTest> showRp $ runP ((wdToken "one" >> wdToken "two") <|>
(wdToken "one" >> wdToken "one"))  (State 5 (tokenize noPos "one two three
four") noPos)
"[\"() [three,four,]\"]"
```

If one would have looked **2** tokens ahead, the first example should have been a success instead of a failure.

The combinator </> chooses alternatives with full lookahead, prefering the left alternative. It first *tries* the left alternative and in case of failure chooses the right alternative:

```
------ Choose with lookahead
{-# INLINE (</>) #-}
(</>) :: Parser st a -> Parser st a -> Parser st a
(</>) f g = try f <|> g
```

Here the "one two three four" experiment succeeds independently of the order of the two parsers.

```
*SAD.Test.ParserTest> showRp $ runP ((wdToken "one" >> wdToken "two") </
> (wdToken "one" >> wdToken "one"))  (State 5 (tokenize noPos "one two three
four") noPos)
"[\"() [three,four,]\"]"
*SAD.Test.ParserTest> showRp $ runP ((wdToken "one" >> wdToken "one") </
> (wdToken "one" >> wdToken "two"))  (State 5 (tokenize noPos "one two three
four") noPos)
"[\"() [three,four,]\"]"
```

9 Parsing a toy natural language

With the tools discussed so far one can implement parsers for simple languages. Standard grammar notation is close to parser definitions in Haskell, using basic parsers and combinators. Our toy language is about birds and fish:

```
sentence = coordination >> fullstop
coordination = (short_sentence >> coordinator >> coordination) </>
short_sentence
short_sentence = noun >> verb
coordinator = wdTokenOf ["and", "or", "but"]
noun = wdTokenOf ["birds", "fish"]
verb = wdTokenOf ["fly", "swim"]
fullstop = smTokenOf "."
```

Note that `coordination` is defined recursively: the recursion is first looking for short sentences with a subsequent coordinator and then calls itself. If that is no longer possible since there is no coordinator left the parser only looks for a short sentence. It is important to use the </> combinator since eventually failure on the left-hand side leads to parsing on the right-hand side. Also note that the logically equivalent definition

```
coordination = (coordination >> coordinator >> short_sentence) </>
short_sentence
```

will lead to non-termination, since the process always takes left-most alternatives; `coordination` is called in a loop until stack overflow. This is an instance of the well-known *left recursion* problem.

Here are some parsing examples for the little language:

```
*SAD.Test.ParserTest> showRp $ runP sentence (State 5 (tokenize noPos "fish
swim.") noPos)
"[\"() []\"]"
*SAD.Test.ParserTest> showRp $ runP sentence (State 5 (tokenize noPos "fish
swim and birds swim.") noPos)
"[\"() []\"]"
*SAD.Test.ParserTest> showRp $ runP sentence (State 5 (tokenize noPos "fish
swim and birds swim and birds fly.") noPos)
"[\"() []\"]"
*SAD.Test.ParserTest> showRp $ runP sentence (State 5 (tokenize noPos "fish
swim and birds swim and fly.") noPos)
":\nunexpected and"
```

The combinator `-|-` instead of `</>` seems to lead to the same results.

For better readability we define a test function in `SAD/Test/ParserTest`:

```
test p s = showRp $ runP p (State 5 (tokenize noPos s) noPos)
```

Then the language examples become:

```
*SAD.Test.ParserTest> test sentence "fish swim and birds swim and birds fly."
"[\"() []\"]"
*SAD.Test.ParserTest> test sentence "fish swim and birds swim and fly."
":\nunexpected and"
...
```

10 More combinators

10.1 Parsing with failures

The `try` parser was already used in the definition of the combinator `</>`:

```
try :: Parser st a -> Parser st a
try p = Parser $ \st ok _ eerr -> runParser p st ok eerr eerr
```

Instead of analyzing the code with its sophisticated treatment of error messages, we just try examples. The following show the effect of try:

```
*SAD.Test.ParserTest> test ((smTokenOf "2" >> smTokenOf "1") <|> (smTokenOf
"2")) "2 swim and birds swim and birds."
":\nunexpected swim"
*SAD.Test.ParserTest> test ((try (smTokenOf "2" >> smTokenOf "1")) <|>
(smTokenOf "2")) "2 swim and birds swim and birds."
"[\"() [swim,and,birds,swim,and,birds,.,]\"]"
```

The ambigious choice operator `-|-` unions up parsing results from both operators:

```
*SAD.Test.ParserTest> test (short_sentence -|- noun) "fish swim and birds swim
and birds fly."
"[\"() [and,birds,swim,and,birds,fly,.,]\",\"() [swim,and,birds,swim,and,birds,
fly,.,]\"]"
```

We obtain a list of parsing results of length > 1. This means that we potentially have ambiguity, which might be resolved in further parsing steps. There might, e.g., be a step where the longest parse is chosen.

10.2 Parsing with options

The parser combinator `opt` allows to throw in an arbitrary result on top of the actual parse result:

```
opt :: a -> Parser st a -> Parser st a
opt x p = p -|- return x

*SAD.Test.ParserTest> test (opt "bird" word) "fish swim."
"[\"\\\"bird\\\" [fish,swim,.,]\",\"\\\"fish\\\" [swim,.,]\"]"
```

This is used in the definition of the Parser `sepBy` which parses separated lists:

```
sepBy :: Parser st a -> Parser st sep -> Parser st [a]
sepBy p sep = liftM2 (:) p $ opt [] $ sep >> sepBy p sep
```

Before analyzing the code we try it out, parsing a sequence of words separated by commas:

```
*SAD.Test.ParserTest> test (sepBy word (smTokenOf ",")) "one,two,three,four"
"[\"[\\\"one\\\",\\\"two\\\",\\\"three\\\",\\\"four\\\"] []\",\"[\\\"one\\\",
\\\"two\\\",\\\"three\\\"] [,,four,]\",\"[\\\"one\\\",\\\"two\\\"] [,,three,,,
four,]\",\"[\\\"one\\\"] [,,two,,,three,,,four,]\"]"
*SAD.Test.ParserTest> test (sepBy word (smTokenOf ",")) "one,two;three,four"
"[\"[\\\"one\\\",\\\"two\\\"] [;,three,,,four,]\",\"[\\\"one\\\"] [,,two,;,
three,,,four,]\"]"
```

So this is an ambigious parser that returns any proper initial segment of works from `"one, two, three, four"`. The parser code with the `$`'s is equivalent to:

```
sepBy p sep = liftM2 (:) p (opt [] (sep >> sepBy p sep))
```

The type of `liftM2` is:

```
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

`liftM2 (:)` lifts the *binary* list operation : onto *two* "monadic" lists (this accounts for the `2` in `liftM2`). Since the type former `[..]` of lists is also a monad, we can examine the operation in the context of lists:

```
*SAD.Test.ParserTest> liftM2 (:) [1,2] [[3,4],[5,6]]
[[1,3,4],[1,5,6],[2,3,4],[2,5,6]]
```

We obtain all combinations `a : b` where `a` is taken from the first list and `b` is taken from the second list (of lists). In Haskell this can be written in a pseudo-settheoretic way where `<-` corresponds to the element relation $\in$.

```
*SAD.Test.ParserTest> [a : b | a <- [1,2], b <- [[3,4],[5,6]]]
[[1,3,4],[1,5,6],[2,3,4],[2,5,6]]
```

The parsing of the comma separated texts like "one, two, three, …" is supposed to produce lists like `["one", "two", "three"]` (in the framework of the `Parser` monad. (Ambiguous) parse results are lists of lists. An application of the parser `word` will produce a singleton list, whose element will be prepended to every list in the list of lists. The ambiguity arise, since the `opt []` can spawn an empty list in every round of the recursion. So the recursive parsing of "one, two, three,four" goes like

```
This has to be CORRECTED

[1:b|b<-[[2:c|c<-[3:d|d<-[[4]]++[[]]]++[[]]][]]++]] =
[1:b|b<-[[]]++[2:c|c<-[[]]++[[3],[3,4]]]]] =
[1:b|b<-[[]]++[[2],[2,3],[2,3,4]]] =
[[1,2,3,4],[1,2,3],[1,2],[1]]
```

where ++ corresponds to the -|- operator which behaves like a union of lists or sets. *to be corrected*.

Why does sepBy include the optional empty sequences? Could we get an unambiguous version without?

```
sepBy' :: Parser st a -> Parser st sep -> Parser st [a]
sepBy' p sep = liftM2 (:) p $ sep >> sepBy' p sep

*SAD.Test.ParserTest> test (sepBy' word (smTokenOf ",")) "one,two,three,four"
":\nunexpected end of input"
```

The problem is that sepBy' is digging deeper and deeper into the input looking for an infinite sequence of separators. The advantage of

```
opt [] (sep >> sepBy p sep)
```

is that it *succeeds* with an empty list when sep >> .. does not find another separator. Parsing texts like "one, two, three, four" is similar to parsing coordinations in the language example above, and it could also be handled similarly.

Leaving away the separators, we get to chains:

```
chain :: Parser st a -> Parser st [a]
chain p = liftM2 (:) p $ opt [] $ chain p

*SAD.Test.ParserTest> test (chain word) "one,two,three,four"
"[\"[\\\"one\\\"] [,,two,,,three,,,four,]\"]"
*SAD.Test.ParserTest> test (chain word) "one two three four"
"[\"[\\\"one\\\",\\\"two\\\",\\\"three\\\",\\\"four\\\"] []\",\"[\\\"one\\\",
\\\"two\\\",\\\"three\\\"] [four,]\",\"[\\\"one\\\",\\\"two\\\"] [three,four,
]\",\"[\\\"one\\\"] [two,three,four,]\"]"
```

10.3 Other options

We can also define options with a "look ahead" of 1 or arbitrarily many tokens:

```
optLL1 :: a -> Parser st a -> Parser st a
optLL1 x p = p <|> return x

optLLx :: a -> Parser st a -> Parser st a
optLLx x p = p </> return x
```

These can be used in seperated lists and in chains. Then LL1 look ahead and the combinator <|> ensure that we need not use the empty list option in the following examples:

```
sepByLL1 :: Parser st a -> Parser st sep -> Parser st [a]
sepByLL1 p sep = liftM2 (:) p $ optLL1 [] $ sep >> sepByLL1 p sep

chainLL1 :: Parser st a -> Parser st [a]
chainLL1 p = liftM2 (:) p $ optLL1 [] $ chainLL1 p

*SAD.Test.ParserTest> test (sepByLL1 word (smTokenOf ",")) "one,two,three,four"
"[\"[\\\"one\\\",\\\"two\\\",\\\"three\\\",\\\"four\\\"] []\"]"
*SAD.Test.ParserTest> test (chainLL1 word) "one two three four 123"
"[\"[\\\"one\\\",\\\"two\\\",\\\"three\\\",\\\"four\\\"] [123,]\"]"
```

10.4 Parsing brackets

Mathematical texts use structuring by all sorts of brackets. This is captured by the following parsers:

```
---- enclosed body (with range)
enclosed :: String -> String -> Parser st a -> Parser st ((SourcePos,
SourcePos), a)
enclosed bg en p = do
  pos1 <- wdTokenPos bg
  x <- p
  pos2 <- wdTokenPos en
  return ((pos1, pos2), x)
```

Note the important and rather intuitive `do`-notation in this definition. The style is similar to classical imperative programming. Commands are performed in order, local variables are instantiated and modified, finally a result is returned in the framework of the encompassing monad.

In this definition, one parses for the initial string `bg` and binds `pos1` to the position of that string; then one parses with `p` and binds the result to `x`; finally one parses for the final string `en` and its position. The result is a combination of the positions and `x`.

```
*SAD.Test.ParserTest> test (enclosed "begin" "end" word) "begin hallo end"
"[\"((,),\\\"hallo\\\") []\"]"
```

The positions of `"begin"` and `"end"` are not visible, since we did not include them into our show functions. Then following are self-explanatory:

```
-- mandatory parentheses, brackets, braces etc.
expar, exbrk, exbrc :: Parser st a -> Parser st a
expar p = snd <$> enclosed "(" ")" p
exbrk p = snd <$> enclosed "[" "]" p
exbrc p = snd <$> enclosed "{" "}" p
```

The `<$>` operator is a monadic operator which lifts a standard function like `snd` (for taking the second component of a pair like `((SourcePos, SourcePos), a)`) into the framework of the right monad `Parser st a`. Intuitively one wants to perform `snd`, type correctness requires "`snd <$>`":

```
*SAD.Test.ParserTest> :t snd
snd :: (a, b) -> b
*SAD.Test.ParserTest> :t \x -> (snd <$> x)
\x -> (snd <$> x) :: Functor f => f (a, b) -> f b
```

One could replace the high level `<$>`-operator by an imperative `do` command:

```
expar' :: Parser st b -> Parser st b
expar' p = do
  x <- enclosed "(" ")" p
  return (snd x)

*SAD.Test.ParserTest> test (expar' word) "(hallo)"
"[\"\\\"hallo\\\" []\"]"
```

Remark: it is often advisable, to code obvious imperative functions in a `do`-format for clarity and readability.

In mathematical formulas, brackets like (...) may be optional:

```
---- optional parentheses
paren :: Parser st a -> Parser st a
paren p = p -|- expar p

*SAD.Test.ParserTest> test (paren word) "hallo"
"[\"\\\"hallo\\\" []\"]"
*SAD.Test.ParserTest> test (paren word) "(hallo)"
```

```
"[\"\\\"hallo\\\" []\"]"
*SAD.Test.ParserTest> test (paren word) "((hallo))"
":\nunexpected ("
```

10.5 Endings of grammatical constructs

ForTheL requires dots or fullstops at the end of statements. These can be enforced by the following parsers.

```
after :: Parser st a -> Parser st b -> Parser st a
after a b = a >>= ((b >>) . return)
```

The purpose of `after a b` is to run the parser `a`, then run `b`, and in case of success of `b` return the result of `a`. Instead of the artful combination of monadic operators, this can be described straightforwardly in `do`-notation:

```
after' :: Monad m => m b -> m a -> m b
after' a b = do
  r <- a
  b
  return r

*SAD.Test.ParserTest> test (after word $ smTokenOf ".") "hallo."
"[\"\\\"hallo\\\" []\"]"
*SAD.Test.ParserTest> test (after word $ smTokenOf ".") "hallo"
":\nunexpected end of input"
*SAD.Test.ParserTest> test (after word $ smTokenOf ".") "hallo.."
"[\"\\\"hallo\\\" [.,]\"]"
*SAD.Test.ParserTest> test (after' word $ smTokenOf ".") "hallo."
"[\"\\\"hallo\\\" []\"]"
*SAD.Test.ParserTest> test (after' word $ smTokenOf ".") "hallo"
":\nunexpected end of input"
*SAD.Test.ParserTest> test (after' word $ smTokenOf ".") "hallo.."
"[\"\\\"hallo\\\" [.,]\"]"
```

As in these examples, expecting a "." after a phrase is a main application of `after`. Parsing for a dot is defined in a peculiar way:

```
---- dot keyword
dot :: Parser st SourceRange
dot = do
  pos1 <- wdTokenPos "." <?> "a dot"
  return $ makeRange (pos1, advancePos pos1 '.')
```

The `<?>`-operator is a custom operator

```
*SAD.Test.ParserTest> :info (<?>)
(<?>) :: Parser st a -> String -> Parser st a
        -- Defined in 'SAD.Parser.Combinators'
```

`parser1 <?> message1` is a parser that first applies `parser1`. If that succeeds, that result is returned. If not, then `message1` is passed to the error system together with position information. With these devices, we shall later see how Naproche enforces dots, e.g., after statements or "Theorem" keywords.

```
---- mandatory finishing dot
finish :: Parser st a -> Parser st a
```

```
finish p = after p dot
```

10.6 Controlling ambiguity

Ambiguity in our kind of parsing means that the list of parse results has more than one element. Here are two ways to handle this:

```
-- Control ambiguity
---- if p is ambiguos, fail and report a well-formedness error

narrow :: Show a => Parser st a -> Parser st a
narrow p = Parser $ \st ok cerr eerr ->
  let pok err eok cok = case eok ++ cok of
        [_] -> ok err eok cok
        ls  ->  eerr $ newErrorMessage (newWfMsg ["ambiguity error" ++ show
(map prResult ls)]) (stPosition st)
  in  runParser p st pok cerr eerr

*SAD.Test.ParserTest> test (narrow (chain word)) "123"
":\nunexpected 123"
*SAD.Test.ParserTest> test (narrow (chain word)) "one 123"
"[\"[\\\"one\\\"] [123,]\"]"
*SAD.Test.ParserTest> test (narrow (chain word)) "one two three 123"
":\nambiguity error[[\"one\",\"two\",\"three\"],[\"one\",\"two\"],[\"one\"]]"
```

Parsing a chain of words is unambiguous, if it has length one; then the result of the chain parser is passed through. Otherwise, an error message is produced with the offending list of parse results.

Another option is to go for longest parse results (which might still be ambiguous).

```
---- only take the longest possible parse, discard all others
takeLongest :: Parser st a -> Parser st a
takeLongest p = Parser $ \st ok cerr eerr ->
  let pok err eok cok
        | null cok  = ok err (longest eok) []
        | otherwise = ok err [] (longest cok)
  in  runParser p st pok cerr eerr
  where
    longest = lng []
    lng ls []          = reverse ls
    lng [] (c:cs)      = lng [c] cs
    lng (l:ls) (c:cs) =
      case compare (stPosition . prState $ l) (stPosition . prState $ c) of
        GT -> lng (l:ls) cs
        LT -> lng [c] cs
        EQ -> lng (c:l:ls) cs

*SAD.Test.ParserTest> test (takeLongest (chain word)) "123"
":\nunexpected 123"
*SAD.Test.ParserTest> test (takeLongest (chain word)) "one 123"
"[\"[\\\"one\\\"] [123,]\"]"
*SAD.Test.ParserTest> test (takeLongest (chain word)) "one two three 123"
"[\"[\\\"one\\\",\\\"two\\\",\\\"three\\\"] [123,]\",\"[\\\"one\\\",
\\\"two\\\"] [three,123,]\",\"[\\\"one\\\"] [two,three,123,]\"]"
```

Observe, that in our examples, `takeLongest` does not work as intended, since we do not have position information available as we have used the `noPos` parameter.

Let us finish our panorama of combinators with “negating” a parse success:

```
---- fail if p succeeds
failing :: Parser st a -> Parser st ()
failing p = Parser $ \st ok cerr eerr ->
  let pok err eok _ =
        if   null eok
        then cerr $ unexpectError (showCurrentToken st) (stPosition st)
        else eerr $ unexpectError (showCurrentToken st) (stPosition st)
      peerr _ = ok (newErrorUnknown (stPosition st)) [PR () st] []
      pcerr _ = ok (newErrorUnknown (stPosition st)) [PR () st] []
  in  runParser p st pok pcerr peerr
  where
    showCurrentToken st = case stInput st of
      (t:ts) -> showToken t
      _      -> "end of input"

*SAD.Test.ParserTest> test (failing word) "one"
":\nunexpected one"
*SAD.Test.ParserTest> test word "one"
"[\"\\\"one\\\" []\"]"
*SAD.Test.ParserTest> test (failing word) "123"
"[\"() [123,]\"]"
*SAD.Test.ParserTest> test (failing word) ""
"[\"() []\"]"
```

11 About monads etc.

There is a lot of theory around monads and similar type classes in functional programming. In these notes we take the practical position that Naproche-SAD is actually programmed monadically and we have to deal with that code. So far, this has been an overhead rather than a help. The monadic context had to be set up, and then simple operations like list operations had to be lifted into that context. Lifting could take place in confusingly many ways that add to the difficulties of the Haskell type system. There are lift operations, seemingly simple operations with a `do` are implicitely lifted, `return` produces lifted values.

Note that our parsers have a user state `st` that so far has been carried around without any gain:

```
-- Continutation passing style ambiguity parser
newtype Parser st a = Parser {runParser :: forall b .
     State st
  -> Continuation st a b
  -> ConsumedFail b
  -> EmptyFail b
  -> b }
```

The advantage of the parser monad will become visible and indispensible when the state `st` will be filled with the ForTheL parser state `FState` for parsing general ForTheL texts. This will be used decisively throughout the parsing process as a large reservoir of information that has been accumulated in the previous parsing process. It can be viewed as a large variable or state that is changing during the course of the computation. This variable, however, is private to the parser and can only be accessed in restricted ways according to some delicate protocol. This is necessary to keep the advantages of functional programming.

Besides the `Parser` monad there are other monads in use, in particular the I/O monad used for file operations or terminal interactions (this is managed automatically for GHCi interactions, but necessary when Naproche runs as a compiled program). Also the list operator `[]` can be viewed as a monad.

12 On the ForTheL statement grammar

We have used combinators for parsing a toy natural language. We shall now look at part of the ForTheL language in similar but more complicated ways. Recall the toy language:

```
sentence = coordination >> fullstop
coordination = (short_sentence >> coordinator >> coordination) </>
short_sentence
short_sentence = noun >> verb
coordinator = wdTokenOf ["and", "or", "but"]
noun = wdTokenOf ["birds", "fish"]
verb = wdTokenOf ["fly", "swim"]
fullstop = smTokenOf "."
```

Complicated sentences are composed from shorter ones and ultimately from `short_sentences` that can be viewed as atomic. Grammar alternatives are expressed by a combinator like `<|>`.

The global statement grammar is described in Andrei Paskevich's *ForTheL Handbook* as follows

constStatement → [the] thesis
| [the] contrary
| [a | an] contradiction

Statements are composed with prepositions and conjunctions as follows:

statement → *headStatement* | *chainStatement*

headStatement → for *quantifiedNotion* { and *quantifiedNotion* } *statement*
| if *statement* then *statement*
| it is wrong that *statement*

chainStatement → *andChain* [and *headStatement*]
| *orChain* [or *headStatement*]
| (*andChain* | *orChain*) iff *statement*

andChain → *primaryStatement* { and *primaryStatement* }

orChain → *primaryStatement* { or *primaryStatement* }

If we suppose for this chapter that primary statements are given, and if we ignore quantified notions for the moment then this is a self-contained grammar similar to our toy example. The corresponding Naproche-SAD code can be found in `SAD.ForTheL.Statement` where we assume for the moment that `simple` statements are the basic building blocks of complicated statements. We have thinned out the Haskell text accordingly

```
statement = headed <|> chained

headed = quStatem <|> ifThenStatem <|> wrongStatem
  where
    quStatem = liftM2 ($) quChain statement
    ifThenStatem = liftM2 Imp
      (markupToken Reports.ifThen "if" >> statement)
      (markupToken Reports.ifThen "then" >> statement)
    wrongStatem =
      mapM_ wdToken ["it", "is", "wrong", "that"] >> fmap Not statement

chained = label "chained statement" $ andOr <|> neitherNor >>= chainEnd
```

```
    where
      andOr = atomic >>= \f -> opt f (andChain f <|> orChain f)
      andChain f =
        fmap (foldl And f) $ and >> atomic `sepBy` and
      orChain f = fmap (foldl Or f) $ or >> atomic `sepBy` or
      and = markupToken Reports.conjunctiveAnd "and"
      or = markupToken Reports.or "or"
      neitherNor = do
        markupToken Reports.neitherNor "neither"; f <- atomic
        markupToken Reports.neitherNor "nor"
        fs <- atomic `sepBy` markupToken Reports.neitherNor "nor"
        return $ foldl1 And $ map Not (f:fs)

  chainEnd f = optLL1 f $ and_st <|> or_st <|> iff_st <|> where_st
    where
      and_st = fmap (And f) $ markupToken Reports.conjunctiveAnd "and" >> headed
      or_st = fmap (Or f) $ markupToken Reports.or "or" >> headed
      iff_st = fmap (Iff f) $ iff >> statement
      where_st = do
        markupTokenOf Reports.whenWhere ["when", "where"]; y <- statement
        return $ foldr zAll (Imp y f) (declNames [] y)

  atomic = label "atomic statement"
    (simple </> (wehve >> thesis))
    where
      wehve = optLL1 () $ wdToken "we" >> wdToken "have"

  thesis = art >> (thes <|> contrary <|> contradiction)
    where
      thes = wdToken "thesis" >> return zThesis
      contrary = wdToken "contrary" >> return (Not zThesis)
      contradiction = wdToken "contradiction" >> return Bot

  simple =  ...
```

For the sake of Isabelle user feedback, “markup tokens” are generated throughout this code. From our perspective, `markupToken` is just an enriched version of `wdToken`; in `SAD.FoThel.Reports` we find:

```
  -- markup tokens while parsing

  markupToken :: Markup.T -> String -> FTL ()
  markupToken markup s = do
    pos <- getPos; wdToken s; addReports $ const [(pos, markup)]

  markupTokenOf :: Markup.T -> [String] -> FTL ()
  markupTokenOf markup ss = do
    pos <- getPos; wdTokenOf ss; addReports $ const [(pos, markup)]
```

We simplify the code for statements by substituting `wdToken` for `markupToken`:

```
  statement = headed <|> chained

  headed = quStatem <|> ifThenStatem <|> wrongStatem
    where
      quStatem = liftM2 ($) quChain statement
      ifThenStatem = liftM2 Imp
```

```
      (wdToken "if" >> statement)
      (wdToken "then" >> statement)
    wrongStatem =
      mapM_ wdToken ["it", "is", "wrong", "that"] >> fmap Not statement

chained = andOr <|> neitherNor >>= chainEnd
  where
    andOr = atomic >>= \f -> opt f (andChain f <|> orChain f)
    andChain f =
      fmap (foldl And f) $ and >> atomic ‘sepBy‘ and
    orChain f = fmap (foldl Or f) $ or >> atomic ‘sepBy‘ or
    and = wdToken "and"
    or = wdToken "or"
    neitherNor = do
      wdToken "neither"; f <- atomic
      d "nor"
      fs <- atomic ‘sepBy‘ wdToken "nor"
      return $ foldl1 And $ map Not (f:fs)

chainEnd f = optLL1 f $ and_st <|> or_st <|> iff_st <|> where_st
  where
    and_st = fmap (And f) $ wdToken "and" >> headed
    or_st = fmap (Or f) $ wdToken "or" >> headed
    iff_st = fmap (Iff f) $ iff >> statement
    where_st = do
      wdTokenOf ["when", "where"]; y <- statement
      return $ foldr zAll (Imp y f) (declNames [] y)

atomic = (simple </> (wehve >> thesis))
  where
    wehve = optLL1 () $ wdToken "we" >> wdToken "have"

thesis = art >> (thes <|> contrary <|> contradiction)
  where
    thes = wdToken "thesis" >> return zThesis
    contrary = wdToken "contrary" >> return (Not zThesis)
    contradiction = wdToken "contradiction" >> return Bot

simple = . . .
```

12.1 Formulas

The task of this parser code is not just to recognize legitimate ForTheL statements, but also to generate correct first-order translations. This is apparent in the types of parsers:

```
*SAD.ForTheL.Statement> :t statement
statement :: Parser FState Formula
*SAD.ForTheL.Statement> :t headed
headed :: Parser FState Formula
*SAD.ForTheL.Statement> :t chained
chained :: Parser FState Formula
*SAD.ForTheL.Statement> :t chainEnd
chainEnd :: Formula -> Parser FState Formula
*SAD.ForTheL.Statement> :t atomic
atomic :: Parser FState Formula
*SAD.ForTheL.Statement> :t thesis
```

```
thesis :: Parser st Formula
*SAD.ForTheL.Statement> :t simple
simple :: Parser FState Formula
```

All these parsers produce formulas, except `chainEnd` which expects a formula and chains it together with the end of the statement. Let us look at the data type of formulas:

```
data Formula =
  All Decl Formula        | Exi Decl Formula |
  Iff Formula Formula     | Imp Formula Formula     |
  Or  Formula Formula     | And Formula Formula     |
  Tag Tag Formula         | Not Formula             |
  Top                     | Bot                     |
  Trm { trName :: String   , trArgs :: [Formula],
        trInfo :: [Formula], trId   :: Int}         |
  Var { trName :: String   , trInfo :: [Formula], trPosition :: SourcePos } |
  Ind { trIndx :: Int, trPosition :: SourcePos }   | ThisT
```

At the "top level" we see quantified and propositionally composed formulas. There are `Top` and `Bot` for true and false. Terms, variables and certain indices are also considered to be formulas in order to have a more uniform buildup of syntax. `ThisT` is a special formula to mark the current thesis.

We can see these components in the grammar/parsing definition:

```
contradiction = wdToken "contradiction" >> return Bot
```

So when the word "contradiction" is encountered in the proper grammatical context, the formula `Bot` is produced. The word "thesis" is marked by a special term which as a term is also a formula. The meaning of that term is only computed at proving time because it depends on the course of the argument up to that position in the text. The coding of this term is very convoluted:

```
thes = wdToken "thesis" >> return zThesis
 ...
zThesis   = zTrm thesisId "#TH#" []
 ...
zTrm :: Int -> String -> [Formula] -> Formula
zTrm tId t ts = Trm t ts [] tId
 ...
thesisId      =  -3 :: Int
```

So the value returned is the formula

```
Trm "#TH#" [] -3
```

This unique marker will be recognized in the reasoning process and instantiated with the right first-order content.

12.2 The parser state

Note that the statement parser also has a mutable state in the data type `FState`:

```
*SAD.ForTheL.Statement> :t statement
statement :: Parser FState Formula
```

This state records information gathered along the parsing process and used locally in the process:

```
data FState = FState {
  adjExpr, verExpr, ntnExpr, sntExpr :: [Prim],
  cfnExpr, rfnExpr, lfnExpr, ifnExpr :: [Prim],
```

```
cprExpr, rprExpr, lprExpr, iprExpr :: [Prim],

tvrExpr :: [TVar], strSyms :: [[String]], varDecl :: [String],
idCount :: Int, hiddenCount :: Int, serialCounter :: Int,
reports :: [Message.Report], pide :: Maybe PIDE }
```

The state holds information about definitions of *adjectives* (`adjExpr`), *verbs* (`verExpr`), ForTheL *notions* (`ntnExpr`) that correspond to noun phrases, and *symbolic notions* (`sntExpr`), and much more bookkeeping information.

12.3 Details of the parser code

We discuss some aspects of the above simplified code where we also ignore quantifiers and quantified notions. First we encounter some grammatical alternatives using `<|>`:

```
statement = headed <|> chained

headed = quStatem <|> ifThenStatem <|> wrongStatem
  where
    quStatem = liftM2 ($) quChain statement
    ifThenStatem = liftM2 Imp
      (wdToken "if" >> statement)
      (wdToken "then" >> statement)
    wrongStatem =
      mapM_ wdToken ["it", "is", "wrong", "that"] >> fmap Not statement
```

In the `ifThenStatem` we want to lift the constructur `Imp` of `Formula` onto the results of the `statement` parsers of the next two lines. Let us look at the operator `liftM2` again:

```
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

We have a binary operation `Imp :: Formula -> Formula -> Formula` on formulas, that is distributed over two monadic results in `m Formula`. Note that parsers produce (monadic) *lists* of parse results, so that all combinations of results have to be admitted. Recall that

```
*SAD.ForTheL.Statement> liftM2 (+) [1,2,3] [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

So `ifThenStatem` produces a list of all implications from parse results of `wdToken "if" >> statement` to parse results of `wdToken "then" >> statement`. This is an adequate first-order interpretation of an "if ... then ..." ForTheL statement.

In `wrongStatem` we use the monadic operator

```
mapM_ :: (Monad m, Foldable t) => (a -> m b) -> t a -> m ()

wdToken :: String -> Parser st ()
```

is a map whose values are parsers that only check that the current token corresponds to the argument string.

```
mapM_ wdToken ["it", "is", "wrong", "that"] :: Parser st ()
```

must thus be equivalent to the obvious parser

```
wdToken "it" >> wdToken "is" >> wdToken "wrong" >> wdToken "that"
```

The subsequent `fmap Not statement` monadically negates all the parse results that `statement` produces. Note the analogy in the list monad:

```
*SAD.ForTheL.Statement> fmap (0 -) [1,2,3]
[-1,-2,-3]
```

This kind of detailed commentary can be continued for the rest of the statement code.

12.4 A propositional mini-ForTheL

To gain a better understanding of the parsing mechanisms we have extracted a grammar and parsing for propositional statements of ForTheL:

```
statement' :: Parser FState Formula
statement' = headed' <|> chained'

headed' = ifThenStatem <|> wrongStatem
  where
    ifThenStatem = liftM2 Imp
      (wdToken "if" >> statement')
      (wdToken "then" >> statement')
    wrongStatem =
      mapM_ wdToken ["it", "is", "wrong", "that"] >> fmap Not statement'

chained' = andOr <|> neitherNor >>= chainEnd'
  where
    andOr = atomic' >>= \f -> opt f (andChain f <|> orChain f)
    andChain f =
      fmap (foldl And f) $ and >> atomic' `sepBy` and
    orChain f = fmap (foldl Or f) $ or >> atomic' `sepBy` or
    and = wdToken "and"
    or = wdToken "or"
    neitherNor = do
      wdToken "neither"; f <- atomic'
      wdToken "nor"
      fs <- atomic' `sepBy` wdToken "nor"
      return $ foldl1 And $ map Not (f:fs)

chainEnd' f = optLL1 f $ and_st <|> or_st <|> iff_st <|> where_st
  where
    and_st = fmap (And f) $ wdToken "and" >> headed'
    or_st = fmap (Or f) $ wdToken "or" >> headed'
    iff_st = fmap (Iff f) $ wdToken "iff" >> statement'
    where_st = do
      wdTokenOf ["when", "where"]
      y <- statement'
      return $ Imp y f

atomic' = (simple' </> (wehve >> thesis'))
  where
    wehve = optLL1 () $ wdToken "we" >> wdToken "have"

thesis' = art' >> (thes <|> contrary <|> contradiction)
  where
    thes = wdToken "thesis" >> return zThesis'
    contrary = wdToken "contrary" >> return (Not zThesis')
    contradiction = wdToken "contradiction" >> return Bot

simple' =  (wdToken "fm1" >> return FM1) <|>
```

```
            (wdToken "fm2" >> return FM2) <|>
            (wdToken "fm3" >> return FM3) <|>
            (wdToken "fm4" >> return FM4) <|>
            (wdToken "fm5" >> return FM5) <|>
            (wdToken "fm6" >> return FM6) <|>
            (wdToken "fm7" >> return FM7) <|>
            (wdToken "fm8" >> return FM8) <|>
            (wdToken "fm9" >> return FM9)

art' = opt () $ wdTokenOf ["a","an","the"]

zThesis' = zTrm' (-3) "#TH#" []

zTrm' :: Int -> String -> [Formula] -> Formula
zTrm' tId t ts = Trm t ts [] tId
```

We import the `Formula` and `FState` data types from proper Naproche-SAD. We have added the propositional constants `FM1` to `FM9` to `Formula` and we have introduced new show instances for them:

```
data Formula =
  All Decl Formula        | Exi Decl Formula |
  Iff Formula Formula     | Imp Formula Formula     |
  Or  Formula Formula     | And Formula Formula     |
  Tag Tag Formula         | Not Formula             |
  Top                     | Bot                     |
  Trm { trName :: String   , trArgs :: [Formula],
        trInfo :: [Formula], trId   :: Int}         |
  Var { trName :: String   , trInfo :: [Formula], trPosition :: SourcePos } |
  Ind { trIndx :: Int, trPosition :: SourcePos }   | ThisT |
  FM1 | FM2 | FM3 | FM4 | FM5 | FM6 | FM7 | FM8 | FM9

showFormula p d = dive
  where
    ...
    dive FM1       = showString "FM1"
    dive FM2       = showString "FM2"
    dive FM3       = showString "FM3"
    dive FM4       = showString "FM4"
    dive FM5       = showString "FM5"
    dive FM6       = showString "FM6"
    dive FM7       = showString "FM7"
    dive FM8       = showString "FM8"
    dive FM9       = showString "FM9"
    ...
```

The `test` function has also been altered to work with a proper element of `FState` although we don't handle the state yet:

```
test p s = showRp $ runP p (State (initFS Nothing) (tokenize noPos s) noPos)
```

where `initFS Nothing :: FState`.

Note that the propositional part of ForTheL is rather small and as yet does not allow many linguistic variants. Experiments show the first-order translations as they would be output by the `-T` option of Naproche-SAD, or error messages:

```
*SAD.Test.ParserTest> test statement' "FM1 and FM2 or FM3"
```

```
"SourcePos {sourceFile = \"\", sourceLine = 0, sourceColumn = 0, sourceOffset =
0, sourceEndOffset = 0}:\nunexpected or"
*SAD.Test.ParserTest> test statement' "FM1 or FM2 and FM3"
"SourcePos {sourceFile = \"\", sourceLine = 0, sourceColumn = 0, sourceOffset =
0, sourceEndOffset = 0}:\nunexpected or"
*SAD.Test.ParserTest> test statement' "FM1 and neither FM2 nor FM3"
"SourcePos {sourceFile = \"\", sourceLine = 0, sourceColumn = 0, sourceOffset =
0, sourceEndOffset = 0}:\nunexpected neither"
*SAD.Test.ParserTest> test statement' "Neither FM1 nor FM2 nor FM3"
"[\"((not FM1 and not FM2) and not FM3) []\",\"(not FM1 and not FM2)
[norSourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn = 0,
sourceOffset = 0, sourceEndOffset = 0},FM3SourcePos {sourceFile = \\\"\\\",
sourceLine = 0, sourceColumn = 0, sourceOffset = 0, sourceEndOffset = 0},]\"]"
*SAD.Test.ParserTest> test statement' "Neither FM1 and FM2 nor FM3"
"SourcePos {sourceFile = \"\", sourceLine = 0, sourceColumn = 0, sourceOffset =
0, sourceEndOffset = 0}:\nunexpected and"
*SAD.Test.ParserTest> test statement' "Neither FM1 nor FM3"
"[\"(not FM1 and not FM3) []\"]"
*SAD.Test.ParserTest> test statement' "FM1 or FM2 or FM3 or FM4"
"[\"(((FM1 or FM2) or FM3) or FM4) []\"]"
```

Mini-ForTheL could be a starting point for many experiments like

- understand the parsing details of mini-ForTheL
- activate the position information and see how error messages point into the code;
- introduce further propositional keywords like "implies" or syntactic sugar like "... holds" and linguistic variants;
- experiment with precedences; should one use commas as natural brackets?
- reintroduce parts of the full Naproche-SAD code, in particular:
- reintroduce quantifiers into the language;
- change the `Formula` data type to include type-theoretic constructs;
- ...

13 `FState` used in parsing "x is equal to y"

Parsers with a state need a way of reading and modifying the state. Recall that

```
data FState = FState {
  adjExpr, verExpr, ntnExpr, sntExpr :: [Prim],
  cfnExpr, rfnExpr, lfnExpr, ifnExpr :: [Prim],
  cprExpr, rprExpr, lprExpr, iprExpr :: [Prim],

  tvrExpr :: [TVar], strSyms :: [[String]], varDecl :: [String],
  idCount :: Int, hiddenCount :: Int, serialCounter :: Int,
  reports :: [Message.Report], pide :: Maybe PIDE }
```

This state is initialized as

```
initFS = FState
  eq [] nt sn
  cf rf [] []
  [] [] [] sp
  [] [] []
```

```
    0 0 0 []
    where
      eq = [
        ([Wd ["equal"], Wd ["to"], Vr], zTrm (-1) "="),
        ([Wd ["nonequal"], Wd ["to"], Vr], Not . zTrm (-1) "=") ]
      sp = [
        ([Sm "="], zTrm (-1) "="),
        ([Sm "!", Sm "="], Not . zTrm (-1) "="),
        ([Sm "-", Sm "<", Sm "-"], zTrm (-2) "iLess"),
        ([Sm "-~-"], \(m:n:_) -> zAll "" $
          Iff (zElem (zVar "") m) (zElem (zVar "") n)) ]
      sn = [ ([Sm "=", Vr], zTrm (-1) "=") ]
      nt = [
        ([Wd ["function","functions"], Nm], zFun . head),
        ([Wd ["set","sets"], Nm], zSet . head),
        ([Wd ["element", "elements"], Nm, Wd ["of"], Vr], \(x:m:_) -> zElem x m),
        ([Wd ["object", "objects"], Nm], zObj . head)]
      rf = [ ([Sm "[", Vr, Sm "]"], \(f:x:_) -> zApp f x)]
      cf = [
        ([Sm "Dom", Sm "(",Vr,Sm ")"], zDom . head),
        ([Sm "(", Vr, Sm ",", Vr, Sm ")"], \(x:y:_) -> zPair x y) ]
```

Most fields initially are "trivial". The non-trivial entries define notations that are hard-coded into Naproche-SAD for sets, functions, ordered pairs etc.

Actually `initFS` is one field short of `FState`, missing `pide :: Maybe PIDE`. Therefore

```
initFS :: Maybe SAD.Core.Message.PIDE -> FState
```

The `Maybe` monad is a data type constructor with the two fields `Just` for a proper value and `Nothing` else. Since we are not addressing the `PIDE` we can simply insert `Nothing` like in our parser testing for `FState` based parsers:

```
test p s = showRp $ runP p (State (initFS Nothing) (tokenize startPos s)
startPos)
```

For full ForTheL parsing, we use (many) parsers with an internal state of data type `FState`.

```
type FTL = Parser FState

*SAD.ForTheL.Statement> :info FTL
type FTL = Parser FState :: * -> *
        -- Defined in ‘SAD.ForTheL.Base’
```

So `FTL` is a pattern that can be instantiated with many data types as target. The main parser for the initial input text is

```
forthel :: FTL [Text]
forthel = section <|> macroOrPretype <|> bracketExpression <|> endOfFile
  where
    section = . . .
 . . .
```

`forthel` produces results of type `[Text]`. Here `Text` is a special and rather complicated format of Naproche-SAD to present the logical content of an input text.

By derivation of types, the alternatives have the same type as forthel:

```
*SAD.ForTheL.Structure> :t bracketExpression
```

```
bracketExpression :: Parser FState [Text]
```

Subparsers "deeper down" also use `FState` for their state, but deliver "simpler" output:

```
*SAD.ForTheL.Structure> :t statement
statement :: Parser FState Formula
```

13.1 Accessing the state

`serialCounter` is used as a "global variable" to number newly occuring variables in a ForTheL text. It is used in `SAD.ForTheL.Base` in the function `makeDecl` which produces an official declaration of a variable whose name `nm` occured at position `pos` in the input:

```
makeDecl :: VarName -> FTL Decl
makeDecl (nm, pos) = do
  serial <- MS.gets serialCounter
  MS.modify (\st -> st {serialCounter = serial + 1})
  return $ Decl nm pos (serial + 1)
```

Obviously the value of `serialCounter` is increased by 1 and then a declaration is formed of this data and returned. To get at the state of the monad we use

```
*SAD.ForTheL.Structure> :info MS.gets
MS.gets :: MS.MonadState s m => (s -> a) -> m a
        -- Defined in ‘Control.Monad.State.Class’
```

This is a general function that was imported from a library file `Control.Monad.State.Class` about monads with states. Given a projection function from the state of the monad into some data type, it returns that value, "decorated" with the monad, or "within" the monad. `serialCounter` is a constructor of `FState` and has the right type:

```
*SAD.ForTheL.Structure> :t serialCounter
serialCounter :: FState -> Int
```

Therefore:

```
*SAD.ForTheL.Structure> :t MS.gets serialCounter
MS.gets serialCounter :: MS.MonadState FState m => m Int
```

Similarly, a modification of the state is possible by `MS.modify`:

```
MS.modify :: MS.MonadState s m => (s -> s) -> m ()
        -- Defined in ‘Control.Monad.State.Class’
```

This function takes a standard state-modifying function and applies it in the setting of a state(ful) monad. Here the state-modifying function is just increasing one component, namely `serialCounter` by 1. Here the function is given by a lambda-expression, and the `{ . . . }` notation allows to address a single field of a data type:

```
*SAD.ForTheL.Structure> :t \st -> st {serialCounter = 1234 + 1}
\st -> st {serialCounter = 1234 + 1} :: FState -> FState
*SAD.ForTheL.Structure> :t MS.modify (\st -> st {serialCounter = 1234 + 1})
MS.modify (\st -> st {serialCounter = 1234 + 1})
  :: MS.MonadState FState m => m ()
```

The output `m ()` signals a successful operation in the type monad, but this information is inside the monad and not released to the outside.

13.2 Declarations

FState has a field varDecl :: [String] which contains names of variables which are “declared” in some sense. There are several operations which access this field. The following simply pulls the field into a list of strings inside the monad, i.e., into FTL [String]

```
getDecl :: FTL [String]
getDecl = MS.gets varDecl
```

Variables can be pretyped, e.g., by some previous typing instruction. Pretyped variables are listed in the FState field tvrExpr :: [TVar] and can be read by

```
getPretyped :: FTL [TVar]
getPretyped = MS.gets tvrExpr
```

The data type TVar is defined as

```
type TVar = ([String], Formula)
```

[What is the explanation of this type?]

addDecl is a parser combinator, probably for some specialized application:

```
addDecl :: [String] -> FTL a -> FTL a
addDecl vs p = do
  dcl <- MS.gets varDecl; MS.modify adv;
  after p $ MS.modify $ sbv dcl
  where
    adv s = s { varDecl = vs ++ varDecl s }
    sbv vs s = s { varDecl = vs }
```

The effect of this parser the application of the parser p with a temporarily enriched list of declared variables: varDecl is read into the monadic variable dcl; the state is modified by adding the list vs to varDecl. Then p is executed, after (after !) which the varDecl is reset to its original value. The special operation MS.modify is applied with the functions adv and sbv dcl.

13.3 Recognizing predicate patterns from FState in the input

The mathematical part of the ForTheL language relies on working with “patterns”. Relations and functions are represented by certain patterns that the parser is supposed to identify and relate to entries in the lists of relations and functions in FState.

In a simple statement like “x is equal to y”, the parser, simplified, should proceed as follows:

- identify the term “x” and put it in a list of entries to be inserted somewhere;
- identify the terminal “is” which means that we deal with a predicate of adjective-form like “blue”, “orthogonal to …”, “equal to …”, etc.;
- identify the pattern “equal to …” and put the term “y” into the list of possible insertions;
- extract the semantics of this pattern out of the list adjExpr in FState;
- fill the insertions into the corresponding “holes” of that semantics.

Patterns for “adjective” predicates are stored in the FState entry adjExpr. In our example we have the pattern “equal to …”, corresponding to the adjExpr entry:

```
([Wd ["equal"], Wd ["to"], Vr], zTrm (-1) "=")
```

Note that the pattern [. . .] in the first position of this ordered pair has type

```
[Wd ["equal"], Wd ["to"], Vr] :: [Patt]
```

where

```
data Patt = Wd [String] | Sm String | Vr | Nm deriving (Eq, Show)
```

The pattern parser has to identify the pattern `[Wd ["equal"], Wd ["to"], Vr]` in the input and output the result `zTrm (-1) "="`.

```
zTrm :: Int -> String -> [Formula] -> Formula
zTrm tId t ts = Trm t ts [] tId
```

So `zTrm (-1) "="` is a function that takes a list of terms like `[x,y, . . . ]` (note that variables are also `Formula`s in our syntax) and turns it into a formula which is supposed to mean "$x = y$":

```
*SAD.Data.Formula.Kit> :t zTrm (-1) "="
zTrm (-1) "=" :: [Formula] -> Formula
```

The process described is implemented in:

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  p <- conjChain doesPredicate
  dig p ts
```

Indeed in our example this reduces to:

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  is -- this parser eats the ["is"]-token with no further effect
  p <- isPredicate
  dig p ts
```

where

```
isPredicate = label "is predicate" $
  pAdj -|- pMultiAdj -|- (with >> hasPredicate)
  where
    pAdj = predicate primAdj
    pMultiAdj = mPredicate primMultiAdj
```

We fall under the alternative `pAdj` and then:

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  is
  p <- predicate primAdj
  dig p ts
```

Since there is no negation in the example statement we take the positive alternative of

```
predicate p = (wdToken "not" >> negative) <|> positive
  where
    positive = do (q, f) <- p term; return $ q . Tag Dig $ f
    negative = do (q, f) <- p term; return $ q . Tag Dig . Not $ f
```

In our case `predicate primAdj` reduces to

```
do
  (q,f) <- primAdj term
  return $ q . Tag Dig $ f
```

Let us assume that `terms` correctly parses the variable "x" and outputs the ordered pair

```
(q, ts) = (id, ["x"])
```

The heart of the matter is the function `primAdj term` whose task it is, in our example, to find the *unary* term corresponding to the function $x \mapsto (x = y)$. Let us computate `primAdj term`. By definition:

```
primAdj = getExpr adjExpr . primPrd
```

This is a *composition* of `getExpr adjExpr` and `primPrd`.

```
getExpr :: (FState -> [a]) -> (a -> FTL b) -> FTL b
getExpr e p = MS.gets e >>=  foldr ((-|-) . try . p ) mzero
```

Then

```
getExpr adjExpr p = MS.gets adjExpr >>=  foldr ((-|-) . try . p ) mzero
```

With `MS.gets adjExpr` we address `FState` and get the momentary content of `adjExpr :: [Prim]` which is a list of entries of type

```
type Prim = ([Patt], [Formula] -> Formula)
```

This is the way how pattern for predicates are saved in `FState`. A predicate is function of type `[Formula] -> Formula` which inserts arguments into a "predicate symbol"; the first component of the ordered pair is the language pattern denoting that predicate. Let us assume informally that the result of `MS.gets adjExpr` is `[prim_1,..,prim_n]`. The `>>=` operator inserts that on the RHS and leads to

```
foldr ((-|-) . try . p ) mzero [prim_1, . . . ,prim_n]
```

`foldr` composes the operation `(-|-) . try . p` over the list and yields

```
(try . p) prim_1 -|-  . . . -|- (try . p) prim_n
```

This is an expression depending on `p`. In our application, `p` is a value of

```
primPrd
  :: Parser st (b1 -> b1, Formula)
     -> ([Patt], [Formula] -> b2) -> Parser st (b1 -> b1, b2)
```

so let us assume that `p = primPrd z` for some variable `z`. Hence

```
primAdj z = (getExpr adjExpr . primPrd) z = getExpr adjExpr (primPrd z)
```

This yields

```
primAdj z = (try . (primPrd z)) prim_1 -|-  . . . -|- (try . (primPrd z)) prim_n
```

We had already seen that at the higher level we have to compute:

```
do
  (q,f) <- primAdj term
  return $ q . Tag Dig $ f
```

Then

```
primAdj term = (try . (primPrd term)) prim_1 -|-  . . . -|- (try . (primPrd term))
prim_n
```

The result is then put into the right format and returned.

Let us try the abstract computation with the "primary" entry for "is equal to". Let us assume that we only have that single entry call `prim_1`. Then we are reduced to

```
primAdj term = try . (primPrd term)) prim_1
```

i.e.,

```
primAdj term = try . (primPrd term)) ([Wd ["equal"], Wd ["to"], Vr], zTrm (-1)
"=")
```

We have the right type:

```
*SAD.Test.ParserTest> :t ((try . (primPrd term)) ([Wd ["equal"], Wd ["to"],
Vr], zTrm (-1) "="))
((try . (primPrd term)) ([Wd ["equal"], Wd ["to"], Vr], zTrm (-1) "="))
  :: Parser FState (Formula -> Formula, Formula)
```

We can see the effect when we put this in the context of `predicate primAdj`, i.e.,

```
*SAD.Test.ParserTest> :t do{(q,f) <- ((try . (primPrd term)) ([Wd ["equal"], Wd
["to"], Vr], zTrm (-1) "=")); return $ q . Tag Dig $ f}
do{(q,f) <- ((try . (primPrd term)) ([Wd ["equal"], Wd ["to"], Vr], zTrm (-1)
"=")); return $ q . Tag Dig $ f}
  :: Parser FState Formula
```

Let us test this parser:

```
*SAD.Test.ParserTest> test (do{(q,f) <- ((try . (primPrd term)) ([Wd ["equal"],
Wd ["to"], Vr], zTrm (-1) "=")); return $ q . Tag Dig $ f}) "equal to z"
"[\"(Dig :: ? = z) []\"]"
```

So the result is a "unary" formula with a question mark, into which the term "x" will be inserted by the `dig` function within

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  is
  p <- predicate primAdj
  dig p ts
```

This convoluted way of accessing the parsing pattern for predicates stored within `FState` is also used for other types of predicates like verbal predicates.

13.4 Plenty of schemas for predicate patterns

Parsing a simple statement like "x is equal to y" is complicated, because we end up with *natural language* parsing. In natural language, the example statement has the form *subject/predicate/(object)*. After parsing "x" there are plenty of possibilities to continue towards a predicate phrase. We go through the various parsers and give natural language examples:

...

...

13.5 Parsing the variables in "x is equal to y"

We go through the (probable) parsing of "x is equal to y". Surely this is a "simple statement":

```
simple = label "simple statement" $ do
```

```
    (q, ts) <- terms; p <- conjChain doesPredicate;
    q' <- optLL1 id quChain;
    -- this part is not in the language description
    -- example: x = y *for every real number x*.
    q . q' <$> dig p ts
```

As we do not have the trailing quantifier chain, this simplifies to:

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  p <- conjChain doesPredicate;
  dig p ts
```

Let us decipher `terms`:

```
terms = label "terms" $
  fmap (foldl1 fld) $ m_term `sepBy` comma
  where
    m_term = quNotion -|- fmap s2m definiteTerm
    s2m (q, t) = (q, [t])
    fld (q, ts) (r, ss) = (q . r, ts ++ ss)
```

For the "simple" term "x" we expect that the parsing is done by `definiteTerm`:

```
definiteTerm = label "definiteTerm" $  symbolicTerm -|- definiteNoun
  where
    definiteNoun = label "definiteNoun" $ paren (art >> primFun term)
```

This leads to `symbolicTerm`:

```
symbolicTerm = fmap ((,) id) sTerm
```

and

```
sTerm :: Parser FState Formula
sTerm = iTerm
  where
    iTerm = lTerm >>= iTl
    iTl t = opt t $ (primIfn sTerm `ap` return t `ap` iTerm) >>= iTl

    lTerm = rTerm -|- label "symbolic term" (primLfn sTerm `ap` lTerm)

    rTerm = cTerm >>= rTl
    rTl t = opt t $ (primRfn sTerm `ap` return t) >>= rTl

    cTerm = label "symbolic term" $ sVar -|- expar sTerm -|- primCfn sTerm
```

Instead of chasing through this code, we can test it:

```
*SAD.Test.ParserTest> test sTerm "x"
"[\"x []\"]"
*SAD.Test.ParserTest> test sTerm "x is equal to y"
"[\"x [is,equal,to,y,]\"]"
```

So this produces the variable `x` which in full is something like

```
Var trName:"x" trInfo:[] trPosition:noPos
```

This is reached up by `symbolicTerm` and `definiteTerm` as

```
(id, Var "x" [] noPos) :: (a -> a, Formula)
```

`m_term` in `terms` reaches this up as

```
(id, [Var "x" [] noPos])
```

This is made into a list of this pair by the chaining operator `‘sepBy‘` into a list of results:

```
[(id, [Var "x" [] noPos])]
```

Then `fmap (foldl fld)` composes all the functions in the list of pairs and concatenates the lists of variables. So the result of `terms` is

```
(id, [Var "x" [] noPos])
```

So this is the result of `terms` in parsing “x is equal to y” by

```
simple = label "simple statement" $ do
  (q, ts) <- terms; p <- conjChain doesPredicate;
  q’ <- optLL1 id quChain;
  -- this part is not in the language description
  -- example: x = y *for every real number x*.
  q . q’ <$> dig p ts
```

We had already seen that “is equal to y” parses to the following formula:

```
*SAD.Test.ParserTest> test (conjChain doesPredicate) "is equal to y"
"[\"(Dig :: ? = y) []\",\"(DigMultiSubject :: ? = !) [toSourcePos {sourceFile
= \\\"\\\", sourceLine = 0, sourceColumn = 0, sourceOffset = 0, sourceEndOffset
= 0},ySourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn = 0,
sourceOffset = 0, sourceEndOffset = 0},]\"]"
```

i.e. to `Dig :: ? = y`. `q’` is trivially set to the identity function `id`. So that `q . q’ = id`.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

is an infix functor that here lifts the identity function into the monad. Since `dig p ts` is already in the monad, `dig p ts` is the result. The (monadic) function `dig` is there to insert the terms from `ts` for question marks in `p`. With the given `Tags` then we (basically) get:

```
dig (Dig :: ? = y) [x] = x = y
```

as expected.

14 Putting information into `FState`

(Global) definition in ForTheL are made on the *top level* of the text. We want to follow some path that leads from a definition in the ForTheL text to a new entry in `adjExpr` within `FState`.

The complete ForTheL input text is parsed by

```
forthel :: FTL [Text]
forthel = section <|> macroOrPretype <|> bracketExpression <|> endOfFile
  where
    section = liftM2 ((:) . TextBlock) topsection forthel
    macroOrPretype = liftM2 (:) (introduceMacro </> pretypeVariable) forthel
    endOfFile = eof >> return []
```

`forthel` produces an internal representation of the input in the format `[Text]` where

```
data Text =
    TextBlock Block
  | TextInstr Instr.Pos Instr
  | NonTextStoredInstr [Instr] -- a way to restore instructions during
verification
  | TextDrop Instr.Pos Instr.Drop
  | TextSynonym SourcePos
  | TextPretyping SourcePos [VarName]
  | TextMacro SourcePos
  | TextError ParseError
  | TextChecked Text
  | TextRoot [Text]

data Block  = Block {
  formula           :: Formula,
  body              :: [Text],
  kind              :: Section,
  declaredVariables :: [Decl],
  name              :: String,
  link              :: [String],
  position          :: SourcePos,
  tokens            :: [Token] }

{- All possible types that a ForThel block can have. -}
data Section =
  Definition | Signature | Axiom       | Theorem | CaseHypothesis  |
  Assumption | Selection | Affirmation | Posit   | LowDefinition
  deriving Eq
```

Imagine a ForTheL definition of the form

> Definition. x is smaller than y iff …

A simplified path through its parsing may look like:

```
topsection = signature <|> definition <|> axiom <|> theorem

definition =
  let define = pretype $ pretypeSentence Posit defExtend defVars noLink
  in  genericTopsection Definition defH define

defExtend = defPredicat -|- defNotion

defPredicat = do
  (f, g) <- wellFormedCheck prdVars defn
  return $ Iff (Tag HeadTerm f) g
  where
    defn = do f <- newPredicat; equiv; g <- statement; return (f,g)
    equiv = iff <|> symbol "<=>"

newPredicat = do n <- newPrdPattern nvr; MS.get >>= addExpr n n True

addExpr :: Formula -> Formula -> Bool -> FState -> FTL Formula
addExpr t@Trm {trName = 'i':'s':' ':_, trArgs = vs} f p st =
  MS.put ns >> return nf
  where
    n = idCount st;
    (pt, nf) = extractWordPattern st (giveId p n t) f
```

```
    fm  = substs nf $ map trName vs
    ns  = st { adjExpr = (pt, fm) : adjExpr st, idCount = incId p n}
```

So finally something like

```
([Wd ["smaller"], Wd ["than"], Vr], zTrm 287 "isSmallerThan")::Prim
```

is added to `adjExpr` and can be used thereafter. The number 287 or so would be generated by the counter for predicates; note that `idcount` is a field (or projection) in the data type `FState`, and `idcount st` returns the value of this counter. `isSmallerThan` is the generated internal symbol for the predicate.

14.1 Pattern extraction

In a definition of "x is smaller than y" the words "smaller" and "than" and the position of the variable y have to be put into a pattern `[Wd ["smaller"], Wd ["than"], Vr]` to generate the above entry for `adjExpr`. Patterns are of data type `[Patt]` where

```
data Patt = Wd [String] | Sm String | Vr | Nm deriving (Eq, Show)
```

From the above chain of functions towards the insertion into `adjExpr` we look for the corresponding pattern by

```
newPredicat = do n <- newPrdPattern nvr; MS.get >>= addExpr n n True

newPrdPattern tvr = multi </> unary </> newSymbPattern tvr
  where
    unary = do
      v <- tvr; (t, vs) <- unaryAdj -|- unaryVerb
      return $ zTrm newId t (v:vs)
    multi = do
      (u,v) <- liftM2 (,) tvr (comma >> tvr);
      (t, vs) <- multiAdj -|- multiVerb
      return $ zTrm newId t (u:v:vs)

    unaryAdj = do is; (t, vs) <- ptHead wlexem tvr; return ("is " ++ t, vs)
    multiAdj = do is; (t, vs) <- ptHead wlexem tvr; return ("mis " ++ t, vs)
    unaryVerb = do (t, vs) <- ptHead wlexem tvr; return ("do " ++ t, vs)
    multiVerb = do (t, vs) <- ptHead wlexem tvr; return ("mdo " ++ t, vs)

ptHead lxm tvr = do
  l <- unwords <$> chain lxm
  (ls, vs) <- opt ([], []) $ ptTail lxm tvr
  return (l ++ ' ' : ls, vs)


ptTail lxm tvr = do
  v <- tvr
  (ls, vs) <- opt ([], []) $ ptHead lxm tvr
  return ("# " ++ ls, v:vs)
```

`ptHead` reads a sequence of "words" before the next variable; `ptTail` reads the variable and returns control over to `ptHead`. Thus in the recognized "word patterns", variables have to be separated by words.

An alternative in `newPrdPattern` is `newSymbPattern` instead of a (unary) word pattern:

```
newSymbPattern tvr = left -|- right
  where
```

```
    left = do
      (t, vs) <- ptHead slexem tvr
      return $ zTrm newId t vs
    right = do
      (t, vs) <- ptTail slexem tvr
      guard $ not $ null $ tail $ words t
      return $ zTrm newId t vs
```

The predicate pattern could start with a variable (`right`) or with a symbol (`left`).

15 Parsing "Every set is equal to y"

We explain the main elements of parsing the complete sentence "Every set is equal to y". Alongside we want to comment the parsing steps "linguistically", including excerpts from Paskevich's grammar.

This parsing starts out like the parsing of "x is equal to y". Again this is a "simple statement":

```
simple = label "simple statement" $ do
  (q, ts) <- terms; p <- conjChain doesPredicate;
  q' <- optLL1 id quChain;
  -- this part is not in the language description
  -- example: x = y *for every real number x*.
  q . q' <$> dig p ts
```

Simple statements apply predicates to terms:

simpleStatement → terms doesPredicate { and doesPredicate }

This corresponds to the familiar noun phrase + verb phrase grammar

As we do not have a *trailing* quantifier chain, this simplifies to:

```
simple = label "simple statement" $ do
  (q, ts) <- terms
  p <- conjChain doesPredicate;
  dig p ts
```

Let us decipher `terms`:

```
terms = label "terms" $
  fmap (foldl1 fld) $ m_term `sepBy` comma
  where
    m_term = quNotion -|- fmap s2m definiteTerm
    s2m (q, t) = (q, [t])
    fld (q, ts) (r, ss) = (q . r, ts ++ ss)
```

terms → term { (, | and) term }

These are lists of single terms where

term → [( ] quantifiedNotion [ )] | definiteTerm

Obviously "Every set" is a quantified notion, and there are no comma separations. Let us decipher `terms`:

```
terms = label "terms" $
  fmap (foldl1 fld) $ m_term `sepBy` comma
  where
    m_term = quNotion -|- fmap s2m definiteTerm
    s2m (q, t) = (q, [t])
    fld (q, ts) (r, ss) = (q . r, ts ++ ss)
```

"Every set" should be parsed by the "every"-alternative of

```
quNotion = label "quantified notion" $
  paren (fa <|> ex <|> no)
  where
    fa = do
      wdTokenOf ["every", "each", "all", "any"]; (q, f, v) <- notion
      vDecl <- mapM makeDecl v
      return (q . flip (foldr dAll) vDecl . blImp f, map pVar v)
    . . .
```

quantifiedNotion → *(every | each | all | any) notion*
| some notion
| no notion

In English, terms can have forms like "For all sets …", where "set" is a common noun, or notion.

The former reduces to

```
do
  wdTokenOf ["every", "each", "all", "any"]
  (q, f, v) <- notion
  vDecl <- mapM makeDecl v
  return (q . flip (foldr dAll) vDecl . blImp f, map pVar v)
```

After parsing the word "every" we are left with "set", being parsed by

```
do
  (q, f, v) <- notion
  vDecl <- mapM makeDecl v
  return (q . flip (foldr dAll) vDecl . blImp f, map pVar v)
```

The central parser in this is

```
notion :: Parser FState (Formula -> Formula, Formula, [(String, SourcePos)])
notion = label "notion" $ gnotion (basentn </> symNotion) stattr >>= digntn
```

notion → *classNoun | classRelation*
classNoun → *{ leftAttribute } primClassNoun [rightAttribute]*
classRelation → *{ leftAttribute } [( ] primClassRelation [ )] [rightAttribute]*

So there in a *classNoun* there is a "central" notion like the `basentn` "set", which could be decorated from to the left with adjectives and with relative sentences and so on to the right. This is organised by the general notion parser `gnotion` using `basentn` for the central notion and `stattr` for the right-hand material.

Since we don't have a symbolic notion, we use the parser

```
gnotion basentn stattr >>= digntn
```

where

```
basentn = fmap digadd $ cm <|> symEqnt <|> (set </> primNtn term)
  where
    cm = wdToken "common" >> primCmNtn term terms
    symEqnt = do
      t <- lexicalCheck isTrm sTerm
      v <- hidden; return (id, zEqu zHole t, [v])
```

leading to the simplification

```
fmap digadd (primNtn term)
```

where

```
primNtn p  = getExpr ntnExpr ntn
  where
    ntn (pt, fm) = do
      (q, vs, ts) <- ntPatt p pt
      return (q, fm $ zHole:ts, vs)
```

Primitive notions can be parametrized notions written in certain patterns with with terms inserted:

pattern → *token* { *token* } [*variable* { *token* { *token* } *variable* }]
token → *small* { *small* }
symbPattern → [*variable*] *symbToken* { *variable symbToken* } [*variable*]
| *word* (*variable* { , *variable* })
| *word* [*variable*]
symbToken → *symbol* { *symbol* }

`getExpr` tries for all the pairs `(pt, fm)` in `ntnExpr` to parse with `ntn (pt, fm)`. Here the fitting pair would be the introduction of "sets" in the initial state of `FState`:

```
([Wd ["set","sets"], Nm], zSet . head)
```

So we parse "set" by

```
ntn ([Wd ["set","sets"], Nm], zSet . head) = do
      (q, vs, ts) <- ntPatt term [Wd ["set","sets"], Nm]
      return (q, zSet . head $ zHole:ts, vs)
```

The parser function `ntPatt` is defined by recursion on the length of the pattern:

```
-- parses a notion: follow the pattern to the name place, record names,
-- then keep following the pattern
ntPatt p (Wd l : ls) = patternWdTokenOf l >> ntPatt p ls
ntPatt p (Nm : ls) = do
  vs <- namlist
  (q, ts) <- wdPatt p ls
  return (q, vs, ts)
ntPatt _ _ = mzero
```

Parsing "set", the word token "set" is swallowed without further effect, and then `ntPatt term [Nm]` is carried out. First

```
namlist = varlist -|- fmap (:[]) hidden
```

goes to the right hand side

```
fmap (:[]) hidden
```

Note that the operator `(:[])` puts an argument inside the square brackets. `fmap` lifts that operator into the monad. We can observe that effect by a test:

```
*SAD.Test.ParserTest> test (fmap (:[]) $ return 123) "tru"
"[\"[123] [truSourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn =
0, sourceOffset = 0, sourceEndOffset = 0},]\"]"
```

So in our case `namlist` will make a singleton list out of the result of

```
hidden = do
  n <- MS.gets hiddenCount
```

```
  MS.modify $ \st -> st {hiddenCount = succ n}
  return ('h':show n, noPos)
```

So the counter hiddenCount in FState is used to form a hidden variable and the counter is increased. Let us test this:

```
*SAD.Test.ParserTest> test hidden "hallo"
"[\"(\\\"h0\\\",SourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn
= 0, sourceOffset = 0, sourceEndOffset = 0}) [halloSourcePos {sourceFile =
\\\"\\\", sourceLine = 0, sourceColumn = 0, sourceOffset = 0, sourceEndOffset =
0},]\"]"
```

Initially, the counter is at 0 and the new hidden variable with the name h0 is formed. To see the increment, we can chain two hidden parsers:

```
*SAD.Test.ParserTest> test (hidden >> hidden) "hallo"
"[\"(\\\"h1\\\",SourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn
= 0, sourceOffset = 0, sourceEndOffset = 0}) [halloSourcePos {sourceFile =
\\\"\\\", sourceLine = 0, sourceColumn = 0, sourceOffset = 0, sourceEndOffset =
0},]\"]"
```

The internal counter is increased to 1 by the first hidden and then the next hidden produces the name h1. namlist puts those results into a singleton list:

```
*SAD.Test.ParserTest> test namlist ""
"[\"[(\\\"h0\\\",SourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn
= 0, sourceOffset = 0, sourceEndOffset = 0})] []\"]"
```

So the result of hidden is a monadic version of [(h123,noPos)].
The next parser is wdPatt term []:

```
-- most basic pattern parser: simply follow the pattern and parse terms with p
-- at variable places
wdPatt p (Wd l : ls) = patternWdTokenOf l >> wdPatt p ls
wdPatt p (Vr : ls) = do
  (r, t) <- p
  (q, ts) <- wdPatt p ls
  return (r . q, t:ts)
wdPatt _ [] = return (id, [])
wdPatt _ _ = mzero
```

So we get return (id, []). In our situation, the parser

```
ntPatt p (Wd l : ls) = patternWdTokenOf l >> ntPatt p ls
ntPatt p (Nm : ls) = do
  vs <- namlist
  (q, ts) <- wdPatt p ls
  return (q, vs, ts)
ntPatt _ _ = mzero
```

returns the monadic triple (id,h123,[]), corresponding to a new hidden variable h123.

```
primNtn p  = getExpr ntnExpr ntn
  where
    ntn (pt, fm) = do
      (q, vs, ts) <- ntPatt p pt
      return (q, fm $ zHole:ts, vs)
```

worked with the pair

```
(pt, fm) = ([Wd ["set","sets"], Nm], zSet . head)
```

The return of `primNtn term` is the triple

```
(id, aSet(?), h123)
```

We now start to assemble together these results of terminal parsers along the sketched parse tree:

The function

```
digadd (q, f, v) = (q, Tag Dig f, v)
```

adds tags, and so

```
fmap digadd (primNtn term)
```

is the monadic version of the triple

```
(id, Tag Dig aSet(?), h123)
```

This would also be the output of the parser `basentn` on input “set”. This parsing is part of:

```
gnotion basentn stattr >>= digntn
```

where

```
gnotion nt ra = do
  ls <- fmap reverse la; (q, f, vs) <- nt;
  rs <- opt [] $ fmap (:[]) $ ra <|> rc
  -- we can use <|> here because every ra in use begins with "such"
  return (q, foldr1 And $ f : ls ++ rs, vs)
  where
    la = opt [] $ liftM2 (:) lc la
    lc = predicate primUnAdj </> mPredicate primMultiUnAdj
    rc = (that >> conjChain doesPredicate <?> "that clause") <|>
      conjChain isPredicate
```

This parser is looking for adjectives and further qualifications by “such that” constructs. This is not the case here. The various extra parsers won’t find new information. Tracing through the definition of `gnotion` shows that in our case `gnotion basentn stattr` gives the same return as `basentn`. This triple is then fed into:

```
digntn (q, f, v) = dig f (map pVar v) >>= \ g -> return (q, g, v)
```

Apart from some typing problem in our triple, which perhaps should be

```
(id, Tag Dig aSet(?), [h123])
```

this means that the variable `h123` will replace the `?` in the formula `Tag Dig ...` The output from

```
gnotion basentn stattr >>= digntn
```

and so should thus be like

```
(id, Tag Dig aSet(h123), [h123])
```

Recall that we are after a quantified notion:

```
quNotion = label "quantified notion" $
  paren (fa <|> ex <|> no)
```

```
    where
      fa = do
        wdTokenOf ["every", "each", "all", "any"]; (q, f, v) <- notion
        vDecl <- mapM makeDecl v
        return (q . flip (foldr dAll) vDecl . blImp f, map pVar v)
      ...
```

On our input "Every set is …" the notion parser follows the above branch

```
gnotion basentn stattr >>= digntn
```

and sets

```
(q, f, v) := (id, Tag Dig aSet(h123), [h123])
```

Using GHCi's `:info` function, we find `makeDecl`:

```
*SAD.Test.ParserTest> :info makeDecl
makeDecl :: VarName -> FTL SAD.Data.Text.Decl.Decl
        -- Defined at /home/koepke/NAPROCHE/Naproche-SAD/src/SAD/ForTheL/
Base.hs:110:1

makeDecl :: VarName -> FTL Decl
makeDecl (nm, pos) = do
  serial <- MS.gets serialCounter
  MS.modify (\st -> st {serialCounter = serial + 1})
  return $ Decl nm pos (serial + 1)
```

Actually `h123` was generated together with its trivial position, and so in `quNotion` we get

```
vDecl := [Decl "h123" noPos serial Number]
```

We are now approaching the return of `quNotion`:

```
return (q . flip (foldr dAll) vDecl . blImp f, map pVar v)
```

`pVar` turns a "variable" of the format "`(name, position)`" into the official variable format in the data type `Formula`. So the second component of the return is basically `[h123]` again.

The first component should be a function of type `Formula -> Formula` which would correspond to the process of putting the quantifier with the quantified notion before a formula. `q` is the identity function, so we have to study

```
flip (foldr dAll) vDecl . blImp f
```

`blImp f` is the function which turns a formula φ into an implication `f ->` φ. `flip (foldr dAll) vDecl` is the function which turns a formula φ into the universal quantification

$\forall$*variables in* `vDecl` φ.

So the return of `quNotion` is basically the pair

($\varphi \mapsto \forall$*variables in* `vDecl` (`f ->` φ), `[h123]`)

where `f` is the formula `Tag Dig aSet(h123)`.

This return is part of

```
terms = label "terms" $
  fmap (foldl1 fld) $ m_term `sepBy` comma
  where
    m_term = quNotion -|- fmap s2m definiteTerm
    s2m (q, t) = (q, [t])
```

```
fld (q, ts) (r, ss) = (q . r, ts ++ ss)
```

where the functions of the `quNotion`'s are composed and the lists of variables are concatenated. So `terms` also outputs

($\varphi \mapsto \forall$*variables in* `vDecl (f ->` φ), `[h123]`)

Again that result is part of

```
simple = label "simple statement" $ do
  (q, ts) <- terms; p <- conjChain doesPredicate;
  q' <- optLL1 id quChain;
  -- this part is not in the language description
  -- example: x = y *for every real number x*.
  q . q' <$> dig p ts
```

So in this `do` sequence:

`(q, ts) :=` ($\varphi \mapsto \forall$*variables in* `vDecl (f ->` φ), `[h123]`)
`p := Dig :: ? = y --` from previous results
`q' := id --` since there is no trailing `quChain`

For the parsing of "is equal to y" we had

```
*SAD.Test.ParserTest> test (conjChain doesPredicate) "is equal to y"
"[\"(Dig :: ? = y) []\",\"(DigMultiSubject :: ? = !) [toSourcePos {sourceFile
= \\\"\\\", sourceLine = 0, sourceColumn = 0, sourceOffset = 0, sourceEndOffset
= 0},ySourcePos {sourceFile = \\\"\\\", sourceLine = 0, sourceColumn = 0,
sourceOffset = 0, sourceEndOffset = 0},]\"]"
```

The output of `simple` is then:

1. "dig" the variable `h123` into `? = y`, replacing `?`;
2. this yields `h123 = y`;
3. put $\forall$`h123 Imp Tag Dig aSet(h123)` in front of `h123 = y`;
4. finally obtain, basically, $\forall$`h123 Imp (Tag Dig aSet(h123)) h123 = y`.

This completes the parsing. The result corresponds to the readable formula $\forall x x = y$, as expected.

16 The ForTheL toplevel

The input text to Naproche-SAD is read by the "global parser" `forthel`:

```
forthel :: FTL [Text]
forthel = section <|> macroOrPretype <|> bracketExpression <|> endOfFile
  where
    section = liftM2 ((:) . TextBlock) topsection forthel
    macroOrPretype = liftM2 (:) (introduceMacro </> pretypeVariable) forthel
    endOfFile = eof >> return []
```

`section` will parse logical sections like definitions or theorems.
`bracketExpression` deals with "system commands" to influence the checking process.
`endOfFile` detects the end of input.
`macroOrPretype` will be discussed soon.

16.1 EOF

The task of `eof` is obvious, but various types and error handling have to be taken care of:

```
---- parse end of input
eof :: Parser st ()
eof = Parser $ \(State st input _) ok _ eerr ->
  case uncons input of
    Nothing -> eerr $ unexpectError "" noPos
    Just (t, ts) ->
      if isEOF t
      then
        let newstate = State st ts (tokenPos t)
            newerr   = newErrorUnknown $ tokenPos t
        in  seq newstate $ ok newerr [] . pure $ PR () newstate
      else eerr $ unexpectError (showToken t) (tokenPos t)
```

where

```
isEOF :: Token -> Bool
isEOF EOF{} = True; isEOF _ = False
```

and the EOF token was set at the end of the token list by

```
tokenize :: SourcePos -> String -> [Token]
tokenize start = posToken start False
  where
    ...
    ...
    posToken pos _ _ = [EOF pos]
```

16.2 Pretyping commands

These are metalevel instructions which fix the (soft) types of certain variables for the subsequent parsing. They typically have the form: "Let x, y stand for sets." or "Let x, y denote sets.". Parsing such instructions is part of the `macroOrPretype` alternative of `forthel`. After parsing the instruction, control is passed back to `forthel` to parse the rest of the text. The pretyping information is kept in the field `tvrExp` of the data type `FState`:

```
data FState = FState {
  ...
  tvrExpr :: [TVar], strSyms :: [[String]], varDecl :: [String],
  ...}

type TVar = ([String], Formula)
```

The entries in `tvrExp` are pairs of *lists* of variables and a formula that expresses the types.

```
pretypeVariable :: Parser FState Text
pretypeVariable = do
  (pos, tv) <- narrow typeVar
  MS.modify $ upd tvx
  return $ TextPretyping pos (fst tv)
  where
    typeVar = do
      pos1 <- getPos; markupToken synonymLet "let"; vs@(_:_) <- varlist;
standFor;
      (g, pos2) <- wellFormedCheck (overfree [] . fst) holedNotion
      let pos = rangePos (pos1, pos2)
      addPretypingReport pos $ map snd vs;
      return (pos, (vs, ignoreNames g))
```

```
    holedNotion = do
      (q, f) <- anotion
      g <- q <$> dig f [zHole]
      (_, pos2) <- dot
      return (g, pos2)

    upd (vs, ntn) st = st { tvrExpr = (map fst vs, ntn) : tvrExpr st }
```

Let us go through the process in “temporal” order.

Determining the current position and part of the `markupToken` are concerned with markup for the IDE and for error handling; `synonymLet` corresponds to some PIDE markup. `markupToken` also parses away the “Let” at the beginning of the instruction.

The list of comma separated variables to be pretyped is obtained by

```
varlist = do
  vs <- var `sepBy` wdToken ","
  nodups $ map fst vs ; return vs
```

`nodups` makes sure that there are no duplicate variable names, otherwise an error message is sent:

```
nodups vs = unless ((null :: [b] -> Bool) $ duplicateNames vs) $
  fail $ "duplicate names: " ++ show vs
```

where

```
{- extracts all duplicateNames (with multiplicity) from a list -}
duplicateNames :: [String] -> [String]
duplicateNames (v:vs) = guardElem vs v `mplus` duplicateNames vs
duplicateNames _      = mzero
```

and

```
guardElem :: [String] -> String -> [String]
guardElem vs v    = guard (v `elem` vs) >> return v
```

So the list of read variables is returned by `varlist` unless the list of duplicate names fails to be the empty list.

Now check the next part of the instruction phrase

```
standFor = wdToken "denote" <|> (wdToken "stand" >> wdToken "for")
```

The next parser is

```
wellFormedCheck (overfree [] . fst) holedNotion
```

where `holedNotion` was defined within `pretypeVariable`. The essential component of `holedNotion` is

```
anotion = label "notion (at most one name)" $
  art >> gnotion basentn rat >>= single >>= hol
  where
    hol (q, f, v) = return (q, subst zHole (fst v) f)
    rat = fmap (Tag Dig) stattr
```

It parses for a `basentn` with all the kind of further left and right attributes that we have seen before. The result is checked to be unary by

```
single (q, f, [v]) = return (q, f, v)
```

```
single _ = fail "inadmissible multinamed notion"
```

If this passes then a question mark `?` is inserted into the formula `f` in place of the single variable `v`. Furthermore the final `dot` is parsed. Then the data is permuted consistent with the types and returned.

16.3 Introducing new notions.

Let us study the parsing of a typical introduction of a new notion like

```
Signature. A real number is a notion.
```

This is handled by the global parser

```
forthel :: FTL [Text]
forthel = section <|> macroOrPretype <|> bracketExpression <|> endOfFile
  where
    section = liftM2 ((:) . TextBlock) topsection forthel
    macroOrPretype = liftM2 (:) (introduceMacro </> pretypeVariable) forthel
    endOfFile = eof >> return []
```

where

```
topsection = signature <|> definition <|> axiom <|> theorem
```

and

```
signature =
  let sigext = pretype $ pretypeSentence Posit sigExtend defVars noLink
  in  genericTopsection Signature sigH sigext
```

The `genericTopsection` deals with various top level sections and parses introductory words like `Signature`, an identifier given to the signature command, assumptions for the signature extension, and finally the "`. . . is a notion`":

```
genericTopsection kind header endparser = do
  pos <- getPos; inp <- getInput; nm <- header;
  toks <- getTokens inp; bs <- body
  let bl = Block.makeBlock zHole bs kind nm [] pos toks
  addBlockReports bl; return bl
  where
    body = assumption <|> endparser
    assumption = topAssume `pretypeBefore` body
    topAssume = pretypeSentence Assumption (asmH >> statement) assumeVars
noLink
```

We apply `genericTopsection` as

```
genericTopsection Signature sigH sigext
```

`Signature` is an element of the datatype `DefType` to indicate a definition of the signature type. `sigH` is looking for a section header with the keyword "Signature":

```
sigH = header ["signature"]
```

where

```
header titles = finish $ markupTokenOf topsectionHeader titles >> optLL1 ""
 topIdentifier
```

This parser looks for the keyword "Signature" and then optionally for an identifier for this section. Thereafter `finish` requires a dot to conclude the header. In our example `Signature. A real number is a notion.` we have successfully parsed away `Signature.` `genericTopsection` then parses `A real number is a notion` with its subparser `body`. Since `A real number is a notion` does not include any assumptions, `body` immediately switches to `sigext`

```
sigext = pretype $ pretypeSentence Posit sigExtend defVars noLink
```

The central parser in this definition is

```
sigExtend = sigPredicat -|- sigNotion
```

We use

```
sigNotion = do
  ((n,h),u) <- wellFormedCheck (ntnVars . fst) sig; uDecl <- makeDecl u
  return $ dAll uDecl $ Imp (Tag HeadTerm n) h
  where
    sig = do
      (n, u) <- newNotion; is; (q, f) <- anotion -|- noInfo
      let v = pVar u; fn = replace v (trm n)
      h <- fmap (fn . q) $ dig f [v]
      return ((n,h),u)

    noInfo =
      art >> wdTokenOf ["notion", "constant"] >> return (id,Top)
    trm Trm {trName = "=", trArgs = [_,t]} = t; trm t = t
```

In our example there are no variables and we take the `noInfo` alternative corresponding to the phrase `is a notion`. The variable `h` will be set to `Top` and `sigNotion` will return a formula like `Imp (Tag HeadTerm n) Top`. We have to find the new notion using

```
newNotion = do
  (n, u) <- newNtnPattern nvr;
  f <- MS.get >>= addExpr n n True
  return (f, u)
```

So `real number` will be recognized as a `newNtnPattern`, the pattern will be created, and `addExpr` will create a new unary predicate symbol `aRealNumber`. Then the pattern and the new symbol will be registered.

16.4 Introducing aliases for statements

In ForTheL it is common to introduce new notations for certain statements or terms. For statements, the "macro" command looks like

> **Let** $x < y$ **stand for** x is smaller than y.

where the left-hand side is a (new) pattern and the right-hand side is a statement about the same variables that are found in the pattern.

This is considered to be a macro definition on the ForTheL toplevel

```
forthel :: FTL [Text]
forthel = section <|> macroOrPretype <|> bracketExpression <|> endOfFile
  where
    section = liftM2 ((:) . TextBlock) topsection forthel
    macroOrPretype = liftM2 (:) (introduceMacro </> pretypeVariable) forthel
    endOfFile = eof >> return []
```

where

```
introduceMacro :: Parser FState Text
introduceMacro = do
  pos1 <- getPos; markupToken macroLet "let"
  (pos2, (f, g)) <- narrow (prd -|- ntn)
  let pos = rangePos (pos1, pos2)
  addMacroReport pos
  MS.get >>= addExpr f (ignoreNames g) False
  return $ TextMacro pos
  where
    prd = wellFormedCheck (prdVars . snd) $ do
      f <- newPrdPattern avr
      standFor; g <- statement; (_, pos2) <- dot; return (pos2, (f, g))
    ntn = wellFormedCheck (funVars . snd) $ do
      (n, u) <- unnamedNotion avr
      standFor; (q, f) <- anotion; (_, pos2) <- dot
      h <- fmap q $ dig f [pVar u]; return (pos2, (n, h))
```

Let us ignore the position handling. This parser first checks for the keyword “let”. Let us assume that we take the `prd` alternative. So after “let” a `newPrdPattern` is parsed until

```
standFor = wdToken "denote" <|> (wdToken "stand" >> wdToken "for")
```

Note that `newPrdPattern` employs the subparser

```
avr = do
  v <- var; guard $ null $ tail $ tail $ fst v
  return $ pVar v
```

which requires that the parsed variables are one-letter words. First

```
var = do
  pos <- getPos
  v <- satisfy (\s -> all isAlphaNum s && isAlpha (head s))
  return ('x':v, pos)
```

identifies a variable name, appends the letter `x` to it and returns that together with position information. The `guard` in `avr` reconstructs the original variable name as `tail $ fst v`. To check that this a one-letter name is equivalent to checking that its `tail` is `null`. If successful the variable is returned in the official `Formula` format by

```
pVar :: (String, SourcePos) -> Formula
pVar (v, pos) = Var v [] pos
```

Let us stress again: to introduce an alias pattern for a predicate, **one-letter** variables $a,b,..,z$ have to be employed, so that one can distinguish them from other works in the pattern.

After this, a `statement` is checked, with a concluding `dot`.

The new pattern `f` and the statement `g` now have to be checked in terms of the occuring variables:

```
prdVars (f, d) | not flat  = return $ "compound expression: " ++ show f
               | otherwise = overfree (free [] f) d
  where
    flat      = isTrm f && allDistinctVars (trArgs f)
```

checks whether the `Formula` `f` is a term and its variables are pairwise distinct. If successful, the next check is carried out by

```
overfree :: [String] -> Formula -> Maybe String
overfree vs f
    | occurs zSlot f = Just $ "too few subjects for an m-predicate " ++ inf
    | not (null sbs) = Just $ "free undeclared variables: "   ++ sbs ++ inf
    | not (null ovl) = Just $ "overlapped variables: "        ++ ovl ++ inf
    | otherwise      = Nothing
  where
    sbs = unwords $ map showVar $ free vs f
    ovl = unwords $ map showVar $ over vs f
    inf = "\n in translation: " ++ show f

    over vs (All v f) = bvrs vs (Decl.name v) f
    over vs (Exi v f) = bvrs vs (Decl.name v) f
    over vs f = foldF (over vs) f

    bvrs vs v f
      | elem v vs = [v]
      | null v    = over vs f
      | otherwise = over (v:vs) f
```

This function compares the list of free variables on the left-hand side with the variables of the right-hand side to check that they basically agree. A more detailed analysis of the rules and errors for variables would be welcome. If the variable requirements are satisfied, the line

```
MS.get >>= addExpr f (ignoreNames g) False
```

feeds the current state into `addExpr`. E.g., in case that the pattern on the left-hand side is and "adjective" pattern, one gets into the following pattern matching alternative:

```
addExpr :: Formula -> Formula -> Bool -> FState -> FTL Formula
addExpr t@Trm {trName = 'i':'s':' ':_, trArgs = vs} f p st =
  MS.put ns >> return nf
  where
    n = idCount st;
    (pt, nf) = extractWordPattern st (giveId p n t) f
    fm  = substs nf $ map trName vs
    ns  = st { adjExpr = (pt, fm) : adjExpr st, idCount = incId p n}
```

So finally the new pattern and the statement in a functional form suitable for variable insertion is then added to the field `adjExpr` in the state.

There are variants of the above procedure if the pattern is of "verb" type, or if one introduces a new notation for a term instead of a statement.

Note 1. One could modify these "macro" commands to achieve certain abbreviations. One could, e.g., fix a group G for the duration of a paragraph, subsection or section and agree: Let $x * y$ stand for $x *^G y$. Such abbreviations can partially replace "notion derivations" to fill in missing or implicit variables.

16.5 Parsing the "Kepler conjecture"

Recall the ForTheL text containing a formulation of the Kepler conjecture close to Tom Hales' original:

```
[synonym number/-s]
Signature. A real number is a notion.
Let x,y stand for real numbers.

Signature. x is greater than y is an atom.
```

```
Signature. A packing of congruent balls
in Euclidean three space is a notion.

Signature. The face centered cubic packing is a packing
of congruent balls in Euclidean three space.

Let P denote a packing of congruent balls in
Euclidean three space.

Signature. The density of P is a real number.

Theorem The_Kepler_conjecture. No packing of congruent
balls in Euclidean three space has density greater than
the density of the face centered cubic packing.
```

By the previous subsections we have an idea of the parsing of the signature extensions. Let us now turn to the Theorem. The header of the Theorem is parsed away by familiar subparsers of `genericTopsection`, and we get to the statement

```
No packing of congruent balls in Euclidean three space has density greater than
the density of the face centered cubic packing.
```

Let us trace through the statement parsing:

```
statement = headed <|> chained
```

The Kepler conjecture is not headed, so we get to

```
chained = label "chained statement" $ andOr <|> neitherNor >>= chainEnd
  where
    andOr = atomic >>= \f -> opt f (andChain f <|> orChain f)
    andChain f =
      fmap (foldl And f) $ and >> atomic ‘sepBy‘ and
    -- we take sepBy here instead of sepByLLx since we do not  know if the
    -- and/or wdToken binds to this statement or to an ambient one
    orChain f = fmap (foldl Or f) $ or >> atomic ‘sepBy‘or
    and = markupToken Reports.conjunctiveAnd "and"
    or = markupToken Reports.or "or"

    neitherNor = do
      markupToken Reports.neitherNor "neither"; f <- atomic
      markupToken Reports.neitherNor "nor"
      fs <- atomic ‘sepBy‘ markupToken Reports.neitherNor "nor"
      return $ foldl1 And $ map Not (f:fs)
```

We get into the `andOr` alternative and there into

```
atomic = label "atomic statement"
  thereIs <|> (simple </> (wehve >> smForm <|> thesis))
  where
    wehve = optLL1 () $ wdToken "we" >> wdToken "have"
```

and

```
simple = label "simple statement" $ do
  (q, ts) <- terms; p <- conjChain doesPredicate;
  q’ <- optLL1 id quChain;
```

```
    -- this part is not in the language description
    -- example: x = y *for every real number x*.
    q . q' <$> dig p ts
```

`terms` will eat the quantified term `No packing of concruent balls in Euclidean three space`, and then `has density greater than the density of the face centered cubic packing.` is dealt with by

```
  doesPredicate = label "does predicate" $
    (does >> (doP -|- multiDoP)) <|> hasP <|> isChain
    where
      doP = predicate primVer
      multiDoP = mPredicate primMultiVer
      hasP = has >> hasPredicate
      isChain = is  >> conjChain (isAPredicat -|- isPredicate)
```

We get into the `hasP` alternative, and `density greater than the density of the face centered cubic packing.` is parsed by

```
  hasPredicate = label "has predicate" $ noPossessive <|> possessive
    where
      possessive = art >> common <|> unary
      unary = fmap (Tag Dig . multExi) $ declared possess `sepBy` (comma >> a
  rt)
      common = wdToken "common" >>
        fmap multExi (fmap digadd (declared possess) `sepBy` comma)

      noPossessive = nUnary -|- nCommon
      nUnary = do
        wdToken "no"; (q, f, v) <- declared possess;
        return $ q . Tag Dig . Not $ foldr mbdExi f v
      nCommon = do
        wdToken "no"; wdToken "common"; (q, f, v) <- declared possess
        return $ q . Not $ foldr mbdExi (Tag Dig f) v
```

Here we follow the `possesive` and the `unary` alternative. The main component of `unary` appears to be the parser

```
  possess = label "possesive notion" $ gnotion (primOfNtn term) stattr >>= digntn
```

where

```
  gnotion nt ra = do
    ls <- fmap reverse la; (q, f, vs) <- nt;
    rs <- opt [] $ fmap (:[]) $ ra <|> rc
    -- we can use <|> here because every ra in use begins with "such"
    return (q, foldr1 And $ f : ls ++ rs, vs)
    where
      la = opt [] $ liftM2 (:) lc la
      lc = predicate primUnAdj </> mPredicate primMultiUnAdj
      rc = (that >> conjChain doesPredicate <?> "that clause") <|>
        conjChain isPredicate
```

...

```
  ...
  ...
  ...
```

17 De Bruijn Variables

Substituting terms into a formula with quantifiers usually requires some "renaming of bound variables" so that variables in the terms do not accidentally get into the range of quantifiers. An elegant method is the de Bruijn notation which always replaces bound variables by integer indices.

(Universally) quantifying a formula by a variable named by a `String` is done by

```
zAll :: String -> Formula -> Formula
zAll v = blAll v . bind v
```

where

```
blAll :: String -> Formula -> Formula
blAll _ Top = Top
blAll v f = All (Decl.nonText v) f
```

and

```
{- bind a variable with name v in a formula.
This also affects any info stored. -}
bind :: String -> Formula -> Formula
bind v = dive 0
  where
    dive n (All u g) = All u $ dive (succ n) g
    dive n (Exi u g) = Exi u $ dive (succ n) g
    dive n Var {trName = u, trPosition = pos}
      | u == v = Ind n pos
    dive n t@Trm{} = t {
      trArgs = map (dive n) $ trArgs t,
      trInfo = map (dive n) $ trInfo t}
    dive _ i@Ind{} = i
    dive n f = mapF (dive n) f
```

`bind v` "dives" into a formula, looking for the string `v` as a name of a (free) variable; along the dive, the depth `n` is increased whenever a quantifier is encountered; if the string is found at depth `n`, the *variable* `Var v` is replaced by the de Bruijn *index* `n`.

Then the function `blAll` puts the quantifier `All` in front of the de Bruijn style formula, with the variable declaration

```
nonText :: String -> Decl
nonText v = Decl v noPos 0
```

So the result of `zAll v f` is the `Formula`

```
All Decl v noPos 0 f'
```

where `f'` is the de Bruijn version of `f`.

Now substitutions can be done straightforwardly without fear of variable capture:

```
{- substitute a formula t for a variable with name v. Does not affect info. -}
subst :: Formula -> String -> Formula -> Formula
subst t v = dive
  where
    dive Var {trName = u} | u == v = t
    dive f = mapF dive f

{- multiple substitutions at the same time. Does not affect info. -}
substs :: Formula -> [String] -> [Formula] -> Formula
```

```
substs f vs ts = dive f
  where
    dive v@Var {trName = u, trInfo = ss} = fromMaybe v (lookup u zvt)
    dive f = mapF dive f
    zvt = zip vs ts
```

Note that `mapF` is a higher-order function which is used to apply a certain function all over a formula:

```
-- Traversing functions
{- map a function over the next structure level of a formula -}
mapF :: (Formula -> Formula) -> Formula -> Formula
mapF fn (All v f) = All v (fn f)
mapF fn (Exi v f) = Exi v (fn f)
mapF fn (Iff f g) = Iff (fn f) (fn g)
mapF fn (Imp f g) = Imp (fn f) (fn g)
mapF fn (Or f g) = Or (fn f) (fn g)
mapF fn (And f g) = And (fn f) (fn g)
mapF fn (Tag a f) = Tag a (fn f)
mapF fn (Not f) = Not (fn f)
mapF fn t@Trm{} = t {trArgs = map fn $ trArgs t}
mapF _ f = f
```

17.1 Boolean simplifications

The definition of `zAll` contained a simplification when quantifying the `Top` formula:

```
blAll _ Top = Top
```

(the `bl...` probably stands for boolean). There are other such simplifications for the other operations of first-order predicate logic:

```
blAnd, blImp :: Formula -> Formula -> Formula
blAnd Top f = f; blAnd (Tag _ Top) f = f
blAnd f Top = f; blAnd f (Tag _ Top) = f
blAnd f g = And f g

blImp _ Top = Top; blImp _ (Tag _ Top) = Top
blImp Top f = f; blImp (Tag _ Top) f = f
blImp f g = Imp f g

blAll, blExi :: String -> Formula -> Formula
blAll _ Top = Top
blAll v f = All (Decl.nonText v) f

blExi _ Top = Top
blExi v f = Exi (Decl.nonText v) f
```

Note that these simplification use classical propositional logic. Such simplifications may become problematic if one wants to feed the results of the parsing into other logics since Naproche-SAD already includes such simplifications in the parsing process. Also these simplifications appear somewhat ad hoc: there is no function `blOr` but there are simplifications for $\vee$ in other places.

17.2 First-order simplifications

Naproche-SAD also employs logical simplifications in quantifier parsing. In the Chapter 2 on ForTheL we saw that the internal translation of

```
No packing of congruent balls in Euclidean three space ...
```

is

```
forall v0 (aPackingOfCongruentBallsInEuclideanThreeSpace(v0) implies not ...
```

This is due to a clause in the code for the natural language "no"-quantifier in the treatment of quantified notions:

```
quNotion = label "quantified notion" $
  paren (fa <|> ex <|> no)
  where
    fa = do ...
    ex = do ...
    no = do
      wdToken "no"; (q, f, v) <- notion
      vDecl<- mapM makeDecl v
      return (q . flip (foldr dAll) vDecl . blImp f . Not, map pVar v)
```

The last line produces an ordered pair whose first component corresponds to a

$$\forall v_0 (P(v_0) \rightarrow \neg ...)$$

operation with a slot where the formula G may be inserted. Such translations go beyond mere parsing and involve first-order processing. If one wants to translate to other logics like type theory, one has to check whether the same transformations are still acceptable.

In the long run it would be cleaner, to exclude logical simplifications from the parsing process and keep the logical translation closer to the original text. Simplifications could then be the first stage of further processing, like proof checking and depend on the background logic.

18 Programming Naproche-SAD

We describe how to run Naproche-SAD, change the source code, recompile it and run the new version of Naproche-SAD on the basis of the easily installed precompiled `tar.gz` distribution provided by Makarius Wenzel at `www.sketis.net`.

Download the current Naproche-SAD package from

```
https://files.sketis.net/Isabelle_Naproche-20190611/
```

We use the `Linux` version

```
https://files.sketis.net/Isabelle_Naproche-20190611/Isabelle_Naproche-
20190611_linux.tar.gz
```

and unpack it somewhere in the file system. This creates a folder `Isabelle_Naproche-20190611` which contains an executable with the same name. Clicking or starting from a terminal opens `Isabelle-jedit` with `Naproche-SAD` available. One can now load `ForTheL` files, ending with `.ftl` and parse and check them.

Isabelle uses a `Naproche-SAD` binary at

```
TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418/x86_64-linux
```

If that does not exist Isabelle searches the binary at

```
~/TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418/.stack-work/
install/x86_64-linux/lts-12.25/8.4.4/bin
```

where it will be put by a `stack build` command issued in the folder

```
TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418
```

We shall use that binary for our experiments. The first run of `stack build` compiles the main program at `.../contrib/naproche-20190418/app/Main.hs` with all submodules at `.../contrib/naproche-20190418/src/SAD`.

From the command line we can run the freshly compiled `Naproche-SAD` by a `stack exec` command:

```
koepke@dell:~/TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418$ stack
exec Naproche-SAD -- examples/powerset.ftl
[Parser] "examples/powerset.ftl"
parsing successful
[Reasoner] "examples/powerset.ftl"
verification started
[Reasoner] "examples/powerset.ftl" (line 23, column 22)
goal:  Take a function f that is defined on M and surjects onto the powerset of
M.
[Reasoner] "examples/powerset.ftl" (line 24, column 1)
goal:  Define N = { x in M | x is not an element of f[x] }.
[Reasoner] "examples/powerset.ftl" (line 25, column 1)
goal:  Then N is not equal to the value of f at any element of M.
[Reasoner] "examples/powerset.ftl" (line 26, column 1)
goal:  Contradiction.
[Export] Error: Bad prover response:
# No SInE strategy applied
# Auto-Mode selected heuristic G_E___207_C18_F1_AE_CS_SP_PI_PS_S0S
# and selection function SelectComplexG.
#
# Presaturation interreduction done
# Proof found!
# SZS status ContradictoryAxioms
```

This is a successful check of `powerset.ftl` which proves that the powerset of a set is strictly larger than the set itself.

To work with the binary built by `stack build` one can hide the preferred `Naproche-SAD` in

```
TEST/Isabelle_Naproche-20190611/contrib/naproche-20190418/x86_64-linux
```

by renaming it, e.g., to `Naproche-SAD.original` and open Isabelle by clicking on the icon in the folder `Isabelle_Naproche-20190611`.

We can change `Naproche-SAD` sources, recompile by `stack build` and restart Isabelle (unfortunately, (re-)starting Isabelle_Naproche with its default settings is quite slow).

19 Modifying the set/class-theory of Naproche-SAD

The present Naproche-SAD allows to reproduce the Russell contradiction in ForTheL:

```
Definition. R = {sets x | x is not an element of x}.

Theorem. Contradiction.
```

This text is readily checked by Naproche-SAD:

```
[Parser] (file "/home/koepke/TEST/Russell.ftl")
parsing successful
[Reasoner] (file "/home/koepke/TEST/Russell.ftl")
```

```
verification started
[Translation] (line 1 of "/home/koepke/TEST/Russell.ftl")
forall v0 ((HeadTerm :: v0 = R) iff (aSet(v0) and forall v1 (aElementOf(v1,v0)
iff (Replacement :: (aSet(v1) and not aElementOf(v1,v1))))))
[Translation] (line 3 of "/home/koepke/TEST/Russell.ftl")
contradiction
[Thesis] (line 3 of "/home/koepke/TEST/Russell.ftl")
thesis: contradiction
[Reasoner] (file "/home/koepke/TEST/Russell.ftl")
verification successful
......
```

The contradiction is due to the automatic registration of `R` as a *set*

```
(HeadTerm :: v0 = R) iff (aSet(v0) and . . .
```

This should be avoided by registering `R` as a *class* instead so that it does not fall under the bounded variable `x` in the definition of `R`. This requires changing the code of Naproche-SAD.

Note. The above situation does *not* mean that Naproche-SAD is outright inconsistent. Rather, Naproche-SAD correspond to working in naive set theory, which allows to form sets $\{x|\,\varphi\}$, and it is up to the author to avoid forming or introducing contradictory sets. Felix Hausdorff wrote in the beginning of *Grundzüge der Mengenlehre*:

> … so wollen wir hier den naiven Mengenbegriff zulassen, dabei aber tatsächlich die Beschränkungen innehalten, die den Weg zu jenem Paradoxon abschneiden.

19.1 An ontology with classes and objects

The notion of "set" as well as some pertinent mechanisms are hard-coded in Naproche-SAD, distributed over several modules. Abstraction terms are automatically registered as sets. We want to use the abstraction mechanism for classes instead. To avoid the Russell contradiction we agree that classes are built from objects and we define that sets are classes that are also objects.

We change the code in three? steps.

19.2 Replacing sets by classes

We go through the code and turn every "set" into "class". We do this by, e.g., saving the original file `SAD.Core.Base.hs` as `SAD.Core.Base.hs.old` and modifying `SAD.Core.Base.hs`.

In the module `SAD.Core.Base` replace `set` by `clss` (since `class` is a Haskell keyword) and `zSet` by `zClass`:

```
-- initial definitions
initialDefinitions = IM.fromList [
  (-1,  equality),
  (-2,  less),
  (-4,  function),
  (-5,  functionApplication),
  (-6,  domain),
  (-7,  clss),
  (-8,  elementOf),
  (-10, pair) ]

equality  = DE [] Top Signature (zEqu (zVar "?0") (zVar "?1")) [] []
less      = DE [] Top Signature (zLess (zVar "?0") (zVar "?1")) [] []
clss     = DE [] Top Signature (zClass $ zVar "?0") [] []
elementOf = DE [zClass $ zVar "?1"] Top Signature
```

```
    (zElem (zVar "?0") (zVar "?1")) [] [[zClass $ zVar "?1"]]
  function  = DE [] Top Signature (zFun $ zVar "?0") [] []
  domain    = DE [zFun $ zVar "?0"] (zClass ThisT) Signature
    (zDom $ zVar "?0") [zClass ThisT] [[zFun $ zVar "?0"]]
  pair      = DE [] Top Signature (zPair (zVar "?0") (zVar "?1")) [] []
  functionApplication =
    DE [zFun $ zVar "?0", zElem (zVar $ "?1") $ zDom $ zVar "?0"] Top Signature
      (zApp (zVar "?0") (zVar "?1")) []
      [[zFun $ zVar "?0"],[zElem (zVar $ "?1") $ zDom $ zVar "?0"]]


  initialGuards = foldr (\f -> DT.insert f True) (DT.empty) [
    zClass $ zVar "?1",
    zFun $ zVar "?0",
    zElem (zVar $ "?1") $ zDom $ zVar "?0"]
```

By these initial definitions, the notion of `class` is given the identifier -7. Furthermore `zSet` is renamed to `zClass`, and certain other notions include that variables in certain places are now classes, like in

```
  elementOf = DE [zClass $ zVar "?1"] Top Signature
    (zElem (zVar "?0") (zVar "?1")) [] [[zClass $ zVar "?1"]]
```

The formula $x \in y$ includes and requires that y is a class.

Now the definition of `zSet` in the module has to become a definition of `zClass`. In `SAD.Data.Formula.Kit` we put classes instead of sets in

```
  -- creation of predefined functions and notions
  zEqu t s  = zTrm equalityId "=" [t,s]
  zLess t s = zTrm lessId "iLess" [t,s]
  zThesis   = zTrm thesisId "#TH#" []
  zFun      = zTrm functionId "aFunction" . pure
  zApp f v  = zTrm applicationId "sdtlbdtrb" [f , v]
  zDom      = zTrm domainId "szDzozmlpdtrp" . pure
  zClass    = zTrm classId "aClass" . pure
  zElem x m = zTrm elementId "aElementOf" [x,m]
  zProd m n = zTrm productId "szPzrzozdlpdtcmdtrp" [m, n]
  zPair x y = zTrm pairId "slpdtcmdtrp" [x,y]
  zObj      = zTrm objectId "aObj" . pure -- this is a dummy for parsing purposes

  -- predefined identifiers

  equalityId    =  -1 :: Int
  lessId        =  -2 :: Int
  thesisId      =  -3 :: Int
  functionId    =  -4 :: Int
  applicationId =  -5 :: Int
  domainId      =  -6 :: Int
  classId       =  -7 :: Int
  elementId     =  -8 :: Int
  productId     =  -9 :: Int
  pairId        = -10 :: Int
  objectId      = -11 :: Int
```

We now have to replace `zSet` by `zClass`. In the initial parser state in the module `SAD.ForTheL.Base` we replace the original assignment of `zSet` to the word pattern for set:

```
  ([Wd ["set","sets"], Nm], zSet . head)
```

by

```
([Wd ["class","classes"], Nm], zClass . head)
```

In the module `SAD.ForTheL.Statement` there are many occurances of “sets”. We have replaced all of them, at a certain risk of loosing some set mechanism for separation or replacement, that we shall have to reconstitute later. Once sets are reintroduced as a notion, something like `zSet` will have to be defined.

Finally `zSet` has to be renamed in the modules `SAD.Core.Reason` and `SAD.Core.ProofTask`.

There are also some set-related functions which have been renamed at the same time.

Now we compile the code with `stack build`. We realize from the errors that a few more renamings are necessary.

Finally we can restart Isabelle_Naproche. The file `powerset.ftl` does no longer check since the system now expects classes instead of sets. Now a modified version of the file checks successfully:

```
[synonym subset/-s] [synonym surject/-s]

Let M denote a class.
Let f denote a function.
Let the value of f at x stand for f[x].
Let f is defined on M stand for Dom(f) = M.
Let the domain of f stand for Dom(f).

Axiom. The value of f at any element of the domain of f is a class.

[synonym subclass/-es]
Definition.
A subclass of M is a class N such that every element of N is an element of M.

Definition.
The powerset of M is the class of subclasses of M.

Definition.
f surjects onto M iff every element of M is equal to the value of f at some
element of the domain of f.

Proposition.
No function that is defined on M surjects onto the powerset of M.
Proof.
Assume the contrary. Take a function f that is defined on M and surjects onto
the powerset of M.
Define N = { x in M | x is not an element of f[x] }.
Then N is not equal to the value of f at any element of M.
Contradiction. qed.
```

19.3 Further tasks

The modification of the hard-coded ontology has to be continued by introducing “objects” as a new notion corresponding to elements of classes. Sets are classes which are objects at the same time.

(Momentarily there is some difficulty with the notion of objects; the word “object” is just a meaningless placeholder, which is internally removed by a function `removeObject`; we shall remove `removeObject`.)

. . .

. . .

. . .