

Advanced Functional Programming

2011-2012, period 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

November 29, 2011

5. Advanced Parser Combinators





[Faculty of Science Information and Computing Sciences]

5.1 Problems with "List of Successes"





[Faculty of Science

Problems with Erroneous Input

▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].

Faculty of Science

Problems with Erroneous Input

- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.



Faculty of Science

Problems with Erroneous Input

- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- ► There is no indication of where things went wrong.



[Faculty of Science

Problems with Space Consumption

- ► A complete result has to be constructed before any part of it is returned
- ► The complete input is present in memory as long as no parse has been found
- ► Efficiency may depend critically on the ordering of the alternatives, and thus on how the grammar was written

For all of these problems we have found solutions.



Faculty of Science

5.2 History Parsers





Replace depth-first by breath-first

We extend the Steps data type so it can hold a computed result (using a GADT and an existential type, to which we will come back later).

data Steps a where

```
\begin{array}{ccc} \mathsf{Step} & :: & \mathsf{Progress} \to \mathsf{Steps} \ \mathsf{a} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{Apply} & :: \forall \mathsf{a} \ \mathsf{b}.(\mathsf{b} \to \mathsf{a}) \ \to \mathsf{Steps} \ \mathsf{b} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{Fail} & :: \dots \end{array}
```

Replace depth-first by breath-first

We extend the Steps data type so it can hold a computed result (using a GADT and an existential type, to which we will come back later).

The Progress field describes how much progress we made in the input (i.e. how much of the input was consumed by this step)

Computing a result

We now adapt our code so we compute a result on the fly, and change the parser type into:

$$\begin{tabular}{ll} \textbf{newtype} & \mathsf{HP} \; \mathsf{st} \; \mathsf{a} \\ & = & \mathsf{HP} \; (\forall \mathsf{r}. (\mathsf{a} \to \mathsf{st} \to \mathsf{Steps} \; \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \; \mathsf{r}) \\ \end{tabular}$$

Faculty of Science

Computing a result

We now adapt our code so we compute a result on the fly, and change the parser type into:

$$\begin{array}{ll} \textbf{newtype} \ \mathsf{HP} \ \mathsf{st} \ \mathsf{a} \\ &= & \mathsf{HP} \ (\forall \mathsf{r}. (\mathsf{a} \to \mathsf{st} \to \mathsf{Steps} \ \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \ \mathsf{r}) \end{array}$$

$$\begin{array}{c}
h \\
p \\
\text{Steps r}
\end{array}$$

best

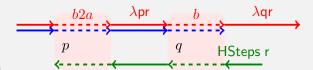
We adapt the function which compares two alternatives.

4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶</p

History parsers are Functor and Applicative

4日 > 4 個 > 4 豆 > 4 豆 > 豆 めの()

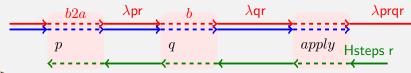
History parsers are Functor and Applicative





History parsers are Functor and Applicative

```
\begin{tabular}{ll} \textbf{instance} & Functor (T st) \textbf{ where} \\ & fmap f (HP ph) & = HP (\lambda k \to ph \ (k \circ f)) \\ \hline \textbf{instance} & Applicative (HP state) \textbf{ where} \\ & HP ph < > \sim (HP qh) \\ & = HP (\lambda k \to ph \ (\lambda pr \to qh \ (\lambda qr \to k \ (pr \ qr)))) \\ & pure a & = HP \ (\$a) \\ \hline \textbf{instance} & Alternative (T state) \textbf{ where} \\ & HP ph < |> HP qh = HP (\lambda k \ inp \to ph \ k \ inp \ `best' \ qh \ k \ inp) \\ & empty & = HP \ (\lambda k \ inp \to noAlts) \\ \hline \end{tabular}
```



5.3 Online Result Construction



Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

イロトイクトイミトイミト ヨ かなべ

Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

▶ for our just introduced parsers this is obvious

We only get the first element of the list of results once q has found a match!



Online results

One of the problems which remains is that we only have access to the result once we have found a complete parse.

- ▶ for our just introduced parsers this is obvious
- but this also holds for the "list-of-successes" method; it is caused by the pattern-matching in the sequential composition

We only get the first element of the list of results once q has found a match!

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

Faculty of Science

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

We observe that, while parsing according to our breadth-first stategy, once we have only **one living alternative left** we could just as well return the result corresponding to the recognised part!



Faculty of Science

Change of Specification

In principle the non-online behaviour is correct: we ask for a complete result, and we can only get a result once we have found at least one complete parse!

We observe that, while parsing according to our breadth-first stategy, once we have only **one living alternative left** we could just as well return the result corresponding to the recognised part!

This is especially useful if we incorporate error-correction in such a way that we are guaranteed to get at least one "possibly successfully corrected" parse.

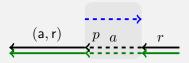
Future Based Parsers

We merge fragments of the result we are constructing with the progress information:

Future Based Parsers

$$\begin{array}{c} \textbf{newtype} \; \mathsf{FP} \; \mathsf{st} \; \mathsf{a} = \mathsf{FP} \; (\forall \mathsf{r}. (\mathsf{st} \to \mathsf{Steps} \quad \mathsf{r}) \to \\ \quad \quad \mathsf{st} \to \mathsf{Steps} \; (\mathsf{a},\mathsf{r}) \\) \end{array}$$

We merge fragments of the result we are constructing with the progress information:



best

We have to make sure that if we compare two alternatives we have progress information at the head:

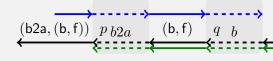
```
\begin{array}{lll} \mathsf{norm} :: \mathsf{Steps} \ \mathsf{a} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{norm} & (\mathsf{Apply} \ \mathsf{f} \ (\mathsf{Step} \ \mathsf{p} \ \mathsf{I} \ )) = & \mathsf{Step} \ \mathsf{p} \ (\mathsf{Apply} \ \mathsf{f} \ \mathsf{I}) \\ \mathsf{norm} & (\mathsf{Apply} \ \mathsf{f} \ (\mathsf{Fail} \ \ldots)) = & \mathsf{Fail} \ \ldots \\ \mathsf{norm} & (\mathsf{Apply} \ \mathsf{f} \ (\mathsf{Apply} \ \mathsf{g} \ \mathsf{I} \ )) = & \mathsf{norm} \ (\mathsf{Apply} \ (\mathsf{f} \circ \mathsf{g}) \ \mathsf{I}) \\ \mathsf{norm} & \mathsf{steps} & = & \mathsf{steps} \\ \mathsf{x} \ `\mathsf{best}` \ \mathsf{y} = \mathsf{norm} \ \mathsf{x} \ `\mathsf{best}'` \ \mathsf{norm} \ \mathsf{y} \\ \mathsf{best}' :: \mathsf{Steps} \ \mathsf{b} \to \mathsf{Steps} \ \mathsf{b} \to \mathsf{Steps} \ \mathsf{b} \end{array}
```

FP is Applicative

```
\begin{tabular}{ll} \textbf{instance} & \mathsf{Applicative} & \mathsf{FP} & \mathsf{qf} & \mathsf{pf} & \mathsf
```

4日 > 4 個 > 4 豆 > 4 豆 > 豆 めの()

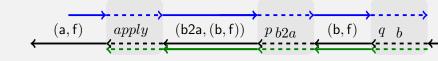
Sequential composition for FParser







Sequential composition for FParser



Faculty of Science

FParser is ISParser

```
pSym \ \mathsf{a} = \mathsf{FP} \ (\lambda \mathsf{k} \ \mathsf{inp} \to \\ \mathbf{case} \ \mathsf{inp} \ \mathsf{of} \ (\mathsf{s} : \mathsf{ss}) \to \mathbf{if} \ \mathsf{s} = \mathsf{a} \ \mathbf{then} \ \mathsf{addStep} \circ \mathsf{push} \ \mathsf{s} \ \mathsf{\$} \ \mathsf{k} \ \mathsf{ss} \\ \mathbf{else} \ \mathsf{Fail} \dots \\ \big[ \ \big] \qquad \to \mathsf{Fail} \dots \\ \big)
```

4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶</p

helper code

```
 \begin{array}{l} \text{eval} :: \mathsf{Steps} \ \ r \to r \\ \text{eval} \ (\mathsf{Step} \ n \ \ l \ ) = (\mathsf{eval} \ l) \\ \text{eval} \ (\mathsf{Fail} \ \ \mathsf{ss} \ \mathsf{ls}) = \ldots \\ \text{eval} \ (\mathsf{Apply} \ \mathsf{f} \ l \ ) = \mathsf{f} \ (\mathsf{eval} \ l) \\ \end{array}
```

```
\begin{array}{ll} \mathsf{push} & :: \mathsf{v} \to \mathsf{Steps} \; \mathsf{r} \to \mathsf{Steps} \; (\mathsf{v},\mathsf{r}) \\ \mathsf{push} \; \mathsf{v} = \mathsf{Apply} \; (\lambda \mathsf{r} \to (\mathsf{v},\mathsf{r})) \end{array}
```

$$\begin{array}{l} \mathsf{apply} :: \mathsf{Steps}\ (\mathsf{b} \to \mathsf{a}, (\mathsf{b}, \mathsf{r})) \to \mathsf{Steps}\ (\mathsf{a}, \mathsf{r}) \\ \mathsf{apply} = \mathsf{Apply}\ (\lambda(\mathsf{b2a}, \sim(\mathsf{b}, \mathsf{r})) \to (\mathsf{b2a}\ \mathsf{b}, \mathsf{r})) \end{array}$$

helper code

```
 \begin{array}{l} \text{eval} :: \mathsf{Steps} \ \ r \to r \\ \text{eval} \ (\mathsf{Step} \ n \ \ l \ ) = (\mathsf{eval} \ l) \\ \text{eval} \ (\mathsf{Fail} \ \ \mathsf{ss} \ \mathsf{ls}) = \dots \\ \text{eval} \ (\mathsf{Apply} \ \mathsf{f} \ l \ ) = \mathsf{f} \ (\mathsf{eval} \ l) \\ \end{array}
```

$$\begin{array}{ll} \mathsf{push} & :: \mathsf{v} \to \mathsf{Steps} \; \mathsf{r} \to \mathsf{Steps} \; (\mathsf{v},\mathsf{r}) \\ \mathsf{push} \; \mathsf{v} = \mathsf{Apply} \; (\lambda \mathsf{r} \to (\mathsf{v},\mathsf{r})) \end{array}$$

apply :: Steps
$$(b \rightarrow a, (b, r)) \rightarrow$$
 Steps (a, r)
apply = Apply $(\lambda(b2a, \sim(b, r)) \rightarrow (b2a \ b, r))$

Notice the \sim in apply. This makes that the function can already produce something!



[Faculty of Science Information and Computing Sciences]

5.4 A Monadic Interface



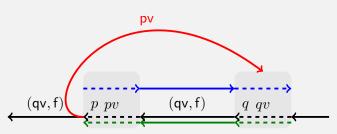


Monadic Interface: Parsing XML

Using a Monadic interface we can e.g. check an XML file for well balanced tags:

Our first attempt" FP

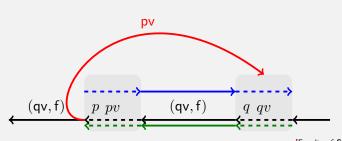
```
instance Monad FP s where \mathbf{p} \ggg \mathbf{q} = \lambda \mathbf{k} \ \mathbf{i} \to \mathbf{let} \ \mathsf{steps} = \mathbf{p} \ (\mathbf{q} \ \underline{pv} \ \mathbf{k}) \ \mathbf{i} \\ (\underline{pv},\_) = \mathsf{eval} \ \mathsf{steps} \\ \mathbf{in} \ \mathsf{Apply} \ \mathsf{snd} \ \mathsf{steps} \\ \mathsf{return} \ \mathbf{v} = pSucceed \ \mathbf{v}
```





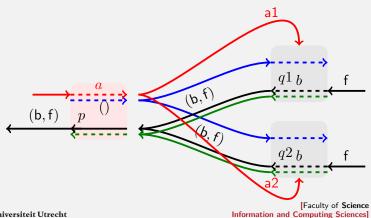
Our first attempt" FP

```
instance Monad FP s where \mathbf{p} \ggg \mathbf{q} = \lambda \mathbf{k} \ \mathbf{i} \to \mathbf{let} \ \mathsf{steps} = \mathbf{p} \ (\mathbf{q} \ \underline{pv} \ \mathbf{k}) \ \mathbf{i} \\ (\underline{pv},\_) = \mathsf{eval} \ \mathsf{steps} \\ \mathbf{in} \ \mathsf{Apply} \ \mathsf{snd} \ \mathsf{steps} \\ \mathsf{return} \ \mathbf{v} = pSucceed \ \mathbf{v}
```



Solution: Combining HP and FP

(>>=) :: HP st a
$$\rightarrow$$
 (a \rightarrow FP st b) \rightarrow FP st b
p >>= q = FP (λ k st \rightarrow p (λ pv st' \rightarrow q pv k st') st)





Making the solution into a Monad

Our next kind of parser is a tupling between a history based and a future based parser:

The Monadic Interface Code

```
\begin{split} & \textbf{instance} \; \mathsf{Monad} \; (Parser \; \mathsf{s}) \; \textbf{where} \\ & (\mathsf{P} \; (\mathsf{HP} \; \mathsf{p}) \; \_) \gg \mathsf{qq} \\ & = \mathsf{P} \; (\mathsf{HP} \; (\lambda \mathsf{k} \; \mathsf{st} \to \mathsf{p} \; (\lambda \mathsf{a} \; \mathsf{st}' \to \mathsf{unHP} \; (\mathsf{qq} \; \mathsf{a}) \; \mathsf{k} \; \mathsf{st}') \; \mathsf{st})) \\ & (\mathsf{FP} \; (\lambda \mathsf{k} \; \mathsf{st} \to \mathsf{p} \; (\lambda \mathsf{a} \; \mathsf{st}' \to \mathsf{unFP} \; (\mathsf{qq} \; \mathsf{a}) \; \mathsf{k} \; \mathsf{st}') \; \mathsf{st})) \\ & \quad \mathsf{where} \; \mathsf{unHP} \; (\mathsf{P} \; (\mathsf{HP} \; \mathsf{h}) \; \_) = \mathsf{h} \\ & \quad \mathsf{unFP} \; (\mathsf{P} \; \_(\mathsf{FP} \; \mathsf{f}) \; ) = \mathsf{f} \\ & \mathsf{return} \; \mathsf{x} = \mathsf{P} \; (pSucceed \; \mathsf{x}) \; (pSucceed \; \mathsf{x}) \end{split}
```

Note that from left hand side of the bind we always take the history based parser, whereas for the right hand side we have two cases to take care of.

Once we have started to tuple various variants of parsers we might just as well:

Once we have started to tuple various variants of parsers we might just as well:

▶ also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway



Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty



Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty
- a list of possible starter symbols to be used in error messages

Once we have started to tuple various variants of parsers we might just as well:

- also tuple a pure recogniser, so we can avoid construction of results which will be discarded anyway
- tuple a possibly empty parser, which is needed for an efficient implementation of the permutation parser with components that may be empty
- a list of possible starter symbols to be used in error messages
- **...**





5.5 Error Correction



Error correction

We can extend the system with an error correcting mechanism.

- we may delete a symbol, at a certain cost
- we may insert a symbol, at a certain cost
- ▶ the function best does not select the longest sequence of steps, but the cheapest
- ▶ limited look-ahead is needed in order to get fast parsers

The correction function pSym

We show a simplified error correcting parser:

```
\begin{array}{ccc} pSym \; \mathbf{a} = \\ & \mathsf{FP} \; \$ \; \mathbf{let} & \mathsf{pSym'} \end{array}
             =\lambdak input 	o
                  case input of
                  inp@(b:bs) \rightarrow if a == b
                                      then Step o push b $ k bs
                                      else Fail o push a $ k bs
                                                             'best'
                                             Fail (pSym' k bs)
                                  \rightarrow Fail \circ push a \$ k input
          in pSym'
```

Universiteit Utrecht

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Refinement of Error-correcting Process

- 1. We may associate a cost with each insertion of deletion step, so we can take the "cheapest future"; some symbols are unlikely to have been forgotten.
- 2. Limited look-ahead in order to speed-up correction process
- 3. Store a report about the corrections taken in the state
- 4. Collect a list of expected symbols, in order to generate nice error messages.
- 5. Use an abstract interpretation to find a non-recursive alternative, in order to avoid infinite insertions.

Computing the minimal length of an alternative

In each tuple which represents a parser we incorporate a value of type Nat:

```
data Nat = Zero
    | Succ Nat deriving Show
\mathsf{nat\_min} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Int} \to (\mathsf{Nat}, \mathsf{Bool})
\begin{array}{lll} \mathsf{nat\_min} \ \_ & \mathsf{Zero} & \_ = (\mathsf{Zero}, \mathsf{False}) \\ \mathsf{nat\_min} \ \mathsf{Zero} & \_ & \_ = (\mathsf{Zero}, \mathsf{True}) \end{array}
nat_min I Infinite _= (I, True)
nat_min (Succ II) (Succ rr) n
    = if n > 1000 then error "problem with comparing length
        else let (v, b) = nat_min | l | rr (n + 1)
                in (Succ v, b))
nat\_add Zero r = r
```

4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶
4□▶</p

 $nat_add (Succ I) r = Succ (nat_add I r))$

The Actual Parser Types

The Actual Parser Types

And the parsing triple:

```
\begin{array}{l} \textbf{data} \; \mathsf{T} \; \mathsf{st} \; \mathsf{a} \\ = \; \mathsf{T} \; - \; \mathsf{history} \\ \; \; (\forall \mathsf{r}. (\mathsf{a} \to \mathsf{st} \to \mathsf{Steps} \; \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \quad \mathsf{r} \; ) \\ \; \; - \; \mathsf{future} \\ \; \; (\forall \mathsf{r}. ( \quad \mathsf{st} \to \mathsf{Steps} \; \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \; (\mathsf{a}, \mathsf{r})) \\ \; \; - \; \mathsf{recogniser} \\ \; \; (\forall \mathsf{r}. ( \quad \mathsf{st} \to \mathsf{Steps} \; \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \quad \mathsf{r} \; ) \end{array}
```

Dealing with Fail

We have been a bit sloppy about failing parsers. We now give the full Fail-alternative of the Steps a type:

```
\begin{tabular}{ll} \textbf{type} \ \mathsf{Syms} = [\mathsf{String}] \\ \textbf{data} \ \mathsf{Steps} \ \mathsf{a} \ \ \textbf{where} \\ \mathsf{Step} \ :: \ \ \mathsf{Progress} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{Apply} :: \ \forall \mathsf{a} \ \mathsf{b}.(\mathsf{b} \to \mathsf{a}) \ \to \mathsf{Steps} \ \mathsf{b} \\ \mathsf{Fail} \ :: \ \ \mathsf{Syms} \ \to [\mathsf{Syms} \to (\mathsf{Int}, \mathsf{Steps} \ \mathsf{a})] \ \to \mathsf{Steps} \ \mathsf{a} \\ \end{tabular}
```

The Strings field keeps track of symbols which were expected.

Dealing with Fail

We have been a bit sloppy about failing parsers. We now give the full Fail-alternative of the Steps a type:

The Strings field keeps track of symbols which were expected. The are collected in the function best

```
\begin{array}{lll} best':: Steps \ b \rightarrow Steps \ b \rightarrow Steps \ b \\ Fail \ sl \ ll & `best'` \ Fail \ sr \ rr = Fail \ (sl \ \# \ sr) \ (ll \ \# \ rr) \\ Fail \ \_ \  \  & `best'` \ Fail \ \_ \  \  = l \\ l & `best'` \ Fail \ \_ \  \  = l \\ \end{array}
```

Evaluating Fail

In case a repair was really necessary the function eval will encounter a Fail in the list of steps:

- 1. all the expected symbols are passed to all the alternatives
- 2. the resulting tree is examined upto a certain depth
- 3. the cheapest branch is taken

Evaluating Fail

In case a repair was really necessary the function eval will encounter a Fail in the list of steps:

- 1. all the expected symbols are passed to all the alternatives
- 2. the resulting tree is examined upto a certain depth
- 3. the cheapest branch is taken

```
 \begin{array}{l} \mathsf{eval} \ \ (\mathsf{Fail} \ \mathsf{expected} \ \mathsf{ls}) \\ = \mathsf{eval} \ (\mathsf{getCheapest} \ 3 \ (\mathsf{map} \ (\$\mathsf{expected}) \ \mathsf{ls})) \\ \mathsf{eval} \dots \end{array}
```

The parsers we have seen thus far cannot deal with ambiguous grammars; the best function does ot find a preferred sequence of Step's.

The parsers we have seen thus far cannot deal with ambiguous grammars; the best function does ot find a preferred sequence of Step's. What should be the result of an ambiguous parser?

The parsers we have seen thus far cannot deal with ambiguous grammars; the best function does ot find a preferred sequence of Step's. What should be the result of an ambiguous parser? We decide to mark ambiguous parsers explicitly:

 $\mathsf{amb} :: \mathsf{P} \mathsf{st} \mathsf{\ a} \to \mathsf{P} \mathsf{\ st} \mathsf{\ [a]}$

The parsers we have seen thus far cannot deal with ambiguous grammars; the best function does ot find a preferred sequence of Step's. What should be the result of an ambiguous parser? We decide to mark ambiguous parsers explicitly:

$$\mathsf{amb} :: \mathsf{P} \mathsf{\,st\,} \mathsf{a} \to \mathsf{P} \mathsf{\,st\,} [\mathsf{a}]$$

Of couse we do not want to parse the parse trailing an ambigous parser more than once.

Placing markes in the Step's

We show the final version of the Step's data type:

```
data Steps a where
       Step :: Progress \rightarrow Steps a
                \rightarrow Steps a
       Apply :: \forall a \ b.(b \rightarrow a) \rightarrow Steps \ b
                 \rightarrow Steps a
       Fail :: Strings \rightarrow [Strings \rightarrow (Cost, Steps a)]
                 \rightarrow Steps a
       Micro :: Int \rightarrow Steps a
                 \rightarrow Steps a
       End_h :: ([a], [a] \rightarrow Steps r) \rightarrow Steps (a, r)
                 \rightarrow Steps (a, r)
       End_f :: [Steps a] \rightarrow Steps a
                 \rightarrow Steps a
```

The "end" markers

- ► The constructors End_h and End_f are used to mark the end of a sequence of steps recognised by an ambiguous parser.
- In case of an ambiguous parser they are encountered at the same time by the best function.
- ▶ In this case the rest of the steps would have been be the same for those alternatives.
- ► Hence we can discard all but one of them, after combining the ambiguous result into a list



The details

```
\begin{array}{ll} \mathsf{amb}\;(\mathsf{P}_-\mathsf{np}\;\mathsf{pl}\;\mathsf{pe}) \\ \ldots ((\mathsf{T}\;\mathsf{ph}\;\mathsf{pf}\;\mathsf{pr}) \\ \mathsf{T}\;(\lambda\mathsf{k}\to \mathsf{removeEnd}_-\mathsf{h}\circ \\ \mathsf{ph}\;(\lambda\mathsf{a}\;\mathsf{st}'\to \mathsf{End}_-\mathsf{h}\;([\mathsf{a}],\lambda\mathsf{as}\to \mathsf{k}\;\mathsf{as}\;\mathsf{st}')\;\mathsf{noAlts})) \\ (\lambda\mathsf{k}\;\mathsf{inp}\to \mathsf{combinevalues}\circ \mathsf{removeEnd}_-\mathsf{f}\;\$ \\ \mathsf{pf}\;(\lambda\mathsf{st}\to \mathsf{End}_-\mathsf{f}\;[\mathsf{k}\;\mathsf{st}]\;\mathsf{noAlts})\;\mathsf{inp}) \\ (\lambda\mathsf{k}\to \mathsf{removeEnd}_-\mathsf{h}\circ \\ \mathsf{pr}\;(\lambda\mathsf{st}'\to \mathsf{End}_-\mathsf{h}\;([\bot],\lambda_-\to \mathsf{k}\;\mathsf{st}')\;\mathsf{noAlts}))) \end{array}
```

The details

```
\begin{array}{ll} \mathsf{amb}\;(\mathsf{P}_-\mathsf{np}\;\mathsf{pl}\;\mathsf{pe}) \\ \dots ((\mathsf{T}\;\mathsf{ph}\;\mathsf{pf}\;\mathsf{pr}) \\ \mathsf{T}\;(\lambda\mathsf{k}\to \mathsf{removeEnd}_-\mathsf{h}\circ \\ \quad \mathsf{ph}\;(\lambda\mathsf{a}\;\mathsf{st}'\to \mathsf{End}_-\mathsf{h}\;([\mathsf{a}],\lambda\mathsf{as}\to \mathsf{k}\;\mathsf{as}\;\mathsf{st}')\;\mathsf{noAlts})) \\ (\lambda\mathsf{k}\;\mathsf{inp}\to \mathsf{combinevalues}\circ \mathsf{removeEnd}_-\mathsf{f}\;\$ \\ \quad \mathsf{pf}\;(\lambda\mathsf{st}\to \mathsf{End}_-\mathsf{f}\;[\mathsf{k}\;\mathsf{st}]\;\mathsf{noAlts})\;\mathsf{inp}) \\ (\lambda\mathsf{k}\to \mathsf{removeEnd}_-\mathsf{h}\circ \\ \quad \mathsf{pr}\;(\lambda\mathsf{st}'\to \mathsf{End}_-\mathsf{h}\;([\bot],\lambda_-\to \mathsf{k}\;\mathsf{st}')\;\mathsf{noAlts}))) \end{array}
```

For the details of the $removeEnd_{-}...$ functions see the "uu-parsinglib" code.

