Polish Parsers, Step by Step

R. John M. Hughes
Chalmers, Gøteborg
Sweden
rjmh@cs.chalmers.se

S. Doaitse Swierstra
Utrecht University
The Netherlands
doaitse@cs.uu.nl

August 26, 2003

Overview

- Motivation
- Recognisers
- Polish Expressions
- Polish Parsing

Overview

- Motivation
- Recognisers
- Polish Expressions
- Polish Parsing
  - Basic parsers
  - Monadic Interface
  - Parsing ambiguous grammars

Overview

- Motivation
- Recognisers
- Polish Expressions
- Polish Parsing
  - Basic parsers
  - Monadic Interface
  - Parsing ambiguous grammars
- Conclusion

Standard parser combinators: Types

$$
\begin{array}{l}
\textbf{infixl}\ 3\ \texttt{<|>} \\
\textbf{infixl}\ 4\ \texttt{<*>}, \texttt{<*}, \texttt{<\$}, \texttt{<\$>}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Parser}\ a & = \mathit{String} \rightarrow [(\mathit{Expr}\ a, \mathit{String})] & \\
(\texttt{<|>}) & :: \quad \mathit{Parser}\ a \rightarrow \mathit{Parser}\ a \rightarrow \mathit{Parser}\ a & \\
(\texttt{<*>}) & :: \quad \mathit{Parser}\ (b \rightarrow a) \rightarrow \mathit{Parser}\ b \rightarrow \mathit{Parser}\ a & \\
\mathit{pSucceed} & :: \quad a \rightarrow \mathit{Parser}\ a & \\
\mathit{pFail} & :: \quad \mathit{Parser}\ a & \\
\mathit{pSym} & :: \quad \mathit{Char} \rightarrow \mathit{Parser}\ a &
\end{array}
$$

The Type $Expr$

- A data type that makes explicit how a value is to be constructed:

$$
\begin{array}{lll}
\textbf{data}\ Expr\ a & = & Val\ a \\
 & \mid & \exists b.\ App\ (Expr\ (b \to a))\ (Expr\ b)
\end{array}
$$

The Type $Expr$

- A data type that makes explicit how a value is to be constructed:

$$
\begin{array}{lll}
\textbf{data}\ Expr\ a & = & Val\ a \\
 & \mid & \exists b.\ App\ (Expr\ (b \to a))\ (Expr\ b)
\end{array}
$$

- An evaluator for such an expression:

$$
\begin{array}{lll}
eval & :: & Expr\ a \to a \\
eval\ (Val\ a) & = & a \\
eval\ (App\ e1\ e2) & = & \textbf{let}\quad f = eval\ e1 \\
 & & \qquad\quad a = eval\ e2 \\
 & & \textbf{in}\quad f\ a
\end{array}
$$

List of Successes Implementation: backtracking

$$
\begin{array}{lll}
pSucceed\ \ v\ \ inp & = & [(v, inp)] \\
pFail & = & [\,] \\
(p\ \texttt{<|>}\ q)\ \ inp & = & p\ inp +\!\!+\ q\ inp \\
(p\ \texttt{<*>}\ q)\ \ inp & = & [\ (App\ f\ \ a, rr) \\
 & & \mid (f, r)\ \ \leftarrow p\ inp \\
 & & ,\ (a, rr)\ \leftarrow q\ r \\
 & &] \\
pSym\ \ a\ \ inp & = & \mathbf{case}\ inp\ \mathbf{of} \\
 & & (s : ss)\ \ \rightarrow\ \mathbf{if}\ \ a \equiv s \\
 & & \qquad\qquad\ \ \mathbf{then}\ [(Val\ s, ss)] \\
 & & \qquad\qquad\ \ \mathbf{else}\ [\,] \\
 & & [\,]\ \ \rightarrow\ [\,]
\end{array}
$$

Shortcomings of this implementation

- *No input is consumed* during parsing

Shortcomings of this implementation

- *No input is consumed* during parsing
  - can be alleviated by explicit extra annotations (e.g. *Parsec*)
  - requires detailed analysis of the grammar
  - has to be done by skilled programmers
  - is error prone

Shortcomings of this implementation

- *No input is consumed* during parsing
  - can be alleviated by explicit extra annotations (e.g. *Parsec*)
  - requires detailed analysis of the grammar
  - has to be done by skilled programmers
  - is error prone
- *No output is produced* during parsing
  - very unfortunate for very long inputs containing similar items
  - complete result is first constructed in memory

Shortcomings of this implementation

- *No input is consumed* during parsing
  - can be alleviated by explicit extra annotations (e.g. *Parsec*)
  - requires detailed analysis of the grammar
  - has to be done by skilled programmers
  - is error prone
- *No output is produced* during parsing
  - very unfortunate for very long inputs containing similar items
  - complete result is first constructed in memory
- Ambiguous grammars are very expensive to parse
  - the part of the input after the ambiguous section is parsed over and over again with the same parsers

Recognisers

- construct a *trace* indicating how the parsing process proceeds
- for the moment we forget about computing a result: **Recognisers**

Recognisers

- construct a *trace* indicating how the parsing process proceeds
- for the moment we forget about computing a result: **Recognisers**
- a **Recogniser** returns a list of *Progress* steps

$$
\begin{array}{lcll}
\mathbf{data}\ Progress & = & Shift\ Progress & \text{— successful parsing step} \\
 & \mid & Done & \text{— successful completion} \\
 & \mid & Fail & \text{— no progress}
\end{array}
$$

Recognisers: continuation style

$$
\begin{array}{lll}
\textbf{newtype}\ \mathit{Rec} & = R & (\mathit{String} \rightarrow \mathit{Progress}) \\
 & \rightarrow & (\mathit{String} \rightarrow \mathit{Progress}) \\
\mathit{rSucceed} & = & R\ (\lambda\ k\ \mathit{input} \rightarrow k\ \mathit{input}) \\
\mathit{rFail} & = & R\ (\lambda\ k\ \mathit{input} \rightarrow \mathit{Fail}) \\
R\ p\ \texttt{<*>}\ R\ q & = & R\ (\lambda\ k\ \mathit{input} \rightarrow p\ (q\ k)\ \mathit{input}) \\
R\ p\ \texttt{<|>}\ R\ q & = & R\ (\lambda\ k\ \mathit{input} \rightarrow p\ k\ \mathit{input} \\
 & & \quad \text{`}\mathit{best}\text{`} \\
 & & \quad q\ k\ \mathit{input}\ ) \\
\mathit{pSym}\ a & = R & (\lambda k\ \mathit{input} \rightarrow \\
 & & \quad \textbf{case}\ \mathit{input}\ \textbf{of} \\
 & & \quad (s : ss) \rightarrow \textbf{if}\ a \equiv s \\
 & & \qquad \textbf{then}\ \mathit{Shift}\ (k\ ss) \\
 & & \qquad \textbf{else}\ \mathit{Fail} \\
 & & \quad [\,] \rightarrow \mathit{Fail} \\
 & &) \\
\mathit{recognize}\ (R\ r)\ \mathit{input} & = r\ \mathit{Done}\ \mathit{input} &
\end{array}
$$

Comparing Alternatives Depth-First

$$
\begin{array}{lll}
l \text{`}best\text{`}\ r = & \textbf{case}\ last\ l\ \textbf{of} & \\
& \quad Fail & \rightarrow\ r \\
& \quad Done & \rightarrow\ \textbf{case}\ last\ r\ \textbf{of} \\
& & \qquad Fail \rightarrow l \\
& & \qquad Done \rightarrow error\ \texttt{"ambiguous grammar"}
\end{array}
$$

$$
\begin{array}{ll}
last\ (Shift\ l) & = last\ \ l \\
last\ Fail & = Fail \\
last\ Done & = Done
\end{array}
$$

Comparing Alternatives Depth-First

$$
\begin{array}{lll}
l \text{`}best\text{`}\ r = & \textbf{case}\ last\ l\ \textbf{of} & \\
& \quad Fail & \rightarrow\ r \\
& \quad Done & \rightarrow\ \textbf{case}\ last\ r\ \textbf{of} \\
& & \qquad Fail \rightarrow l \\
& & \qquad Done \rightarrow error\ \texttt{"ambiguous grammar"}
\end{array}
$$

$$
\begin{array}{ll}
last\ (Shift\ l) & = last\ \ l \\
last\ Fail & = Fail \\
last\ Done & = Done
\end{array}
$$

- first the left alternative is completely evaluated, forced by the call of *last*
- and subsequently the right one

Comparing Alternatives Breadth-First

$$
\begin{array}{lll}
Fail & \text{`}best\text{`}\ p & = p \\
q & \text{`}best\text{`}\ Fail & = q \\
Done & \text{`}best\text{`}\ Done & = error\ \texttt{"ambiguous grammar"} \\
Done & \text{`}best\text{`}\ q & = Done \\
p & \text{`}best\text{`}\ Done & = Done \\
(Shift\ v) & \text{`}best\text{`}\ (Shift\ w) & = Shift\ (v\ \text{`}best\text{`}\ w)
\end{array}
$$

Comparing Alternatives Breadth-First

$$
\begin{array}{lll}
Fail & \text{`}best\text{`}\ p & = p \\
q & \text{`}best\text{`}\ Fail & = q \\
Done & \text{`}best\text{`}\ Done & = error\ \texttt{"ambiguous grammar"} \\
Done & \text{`}best\text{`}\ q & = Done \\
p & \text{`}best\text{`}\ Done & = Done \\
(Shift\ v) & \text{`}best\text{`}\ (Shift\ w) & = Shift\ (v\ \text{`}best\text{`}\ w)
\end{array}
$$

- the last alternative makes the comparison breadth-first
- the uses of $best$ actually drive the parsing process, by asking recognisers to produce $Progess$ to be compared
- if both alternatives have recognised a symbol ($Shift$) we do not choose, but just pass on the information that a symbol was succesfully recognised
- observable behaviour is the same as depth-first
- all alternatives are explored "in parallel"

From Recognisers to Parsers

- each time we recognise a token we have a fragment of the final result at our hands
- together these constitute the overall result

From Recognisers to Parsers

- each time we recognise a token we have a fragment of the final result at our hands
- together these constitute the overall result
- *provided we now how to combine them*

From Recognisers to Parsers

- each time we recognise a token we have a fragment of the final result at our hands
- together these constitute the overall result
- *provided we now how to combine them*
- so we let ourselves be inspired by Polish notation

The Polish Data Type

- We extend the type with a continuation parameter, so

$$\begin{array}{lll} \textbf{data}\ \mathit{Expr}\ a & = & \mathit{Val}\ a \\ & \mid & \exists b.\ \mathit{App}\ (\mathit{Expr}\ (b \to a))\ (\mathit{Expr}\ b) \end{array}$$

The Polish Data Type

- We extend the type with a continuation parameter, so

$$\begin{array}{lll} \textbf{data}\ Expr\ a & = & Val\ a \\ & \mid & \exists b.\ App\ (Expr\ (b \to a))\ (Expr\ b) \end{array}$$

- becomes

$$\begin{array}{lll} \textbf{data}\ Polish\ a\ s & = & Val\ a\ s \\ & \mid & \exists b.\ App\ (Polish\ (b \to a)\ (Polish\ b\ s)) \end{array}$$

The Polish Data Type

- We extend the type with a continuation parameter, so

$$\begin{array}{lll} \textbf{data}\ Expr\ a & = & Val\ a \\ & \mid & \exists b.\ App\ (Expr\ (b \to a))\ (Expr\ b) \end{array}$$

- becomes

$$\begin{array}{lll} \textbf{data}\ Polish\ a\ s & = & Val\ a\ s \\ & \mid & \exists b.\ App\ (Polish\ (b \to a)\ (Polish\ b\ s)) \end{array}$$

- The corresponding $eval$ function :

$$\begin{array}{lll} eval\ :: Polish & & a\ s \to (a,\ s) \\ eval\ (Val\ a\ \ s) & = & (a, s) \\ eval\ (App\ \ \ s) & = & \textbf{let}\ (f,\ \ ss) = eval\ s \\ & & \quad\ \ (a,\ \ sss) = eval\ ss \\ & & \textbf{in}\ \ (f\ a,\ sss) \end{array}$$

Interleaving Polish and Progress

- Parsers will now return an interleaving of $Progress$ and $Polish$ steps:

Interleaving Polish and Progress

- Parsers will now return an interleaving of $Progress$ and $Polish$ steps:

$$
\begin{array}{lcll}
\textbf{data}\ Steps\ a\ s = & & Val & a\ s \\
\mid & \exists b. & App & (Steps\ (b \rightarrow a)\ (Steps\ b\ s)) \\
\mid & & Shift & (Steps\ a\ s) \\
\mid & & Done & (Steps\ a\ s) \\
\mid & & Fail &
\end{array}
$$

Interleaving Polish and Progress

- Parsers will now return an interleaving of *Progress* and *Polish* steps:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Steps}\ a\ s\ = & & \mathit{Val}\quad a\ s \\
\quad\mid & \exists b. & \mathit{App}\quad (\mathit{Steps}\ (b \to a)\ (\mathit{Steps}\ b\ s)) \\
\quad\mid & & \mathit{Shift}\ (\mathit{Steps}\ a\ s) \\
\quad\mid & & \mathit{Done}\ (\mathit{Steps}\ a\ s) \\
\quad\mid & & \mathit{Fail}
\end{array}
$$

- and the evaluator ignores *Progress* steps

$$
\begin{array}{lll}
\mathit{evalSteps}\ (\mathit{Val}\ a\ \ s) & = & (a, s) \\
\mathit{evalSteps}\ (\mathit{App}\ \ \ s) & = & \textbf{let}\ (f, s') \ = \mathit{evalSteps}\ s \\
 & & \qquad\ \ (a, s'') \ = \mathit{evalSteps}\ s' \\
 & & \textbf{in}\ \ (f\ a, s'') \\
\mathit{evalSteps}\ (\mathit{Shift}\ \ v) & = & \mathit{evalSteps}\ v \\
\mathit{evalSteps}\ (\mathit{Done}\ \ v) & = & \mathit{evalSteps}\ v \\
\mathit{evalSteps}\ \mathit{Fail} & = & \mathit{error}\ \texttt{"wrong input"}
\end{array}
$$

Parsers

$$
\begin{array}{l}
pSucceed\ \ a \qquad\qquad = P\ (Val\ a\cdot) \\
P\ p \quad \texttt{<*>}\ P\ \ q \quad = P\ ((App\cdot\ )\cdot p \quad \cdot q) \\
pSym \quad a \\
\quad = P \quad (\lambda \quad k\ \ input \rightarrow \\
\qquad\qquad\qquad \textbf{case}\ input\ \textbf{of} \\
\qquad\qquad\qquad (s : ss) \ \rightarrow \quad \textbf{if} \quad a \equiv s \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (Shift \cdot Val\ s \cdot k)\ \ ss \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ Fail \\
\qquad\qquad\qquad [\,] \qquad \rightarrow \quad Fail \\
\qquad\qquad\quad) \\
parse\ p\ input \\
\quad = \ \textbf{let} \qquad (result, rest) \qquad = \\
\qquad\qquad\qquad evalSteps\ \ (p\ \ (\lambda rest \rightarrow Done\ (Val\ rest\ ())) \\
\qquad\qquad\qquad\qquad\qquad\qquad input \\
\qquad\qquad\qquad\qquad\qquad\quad) \\
\qquad \textbf{in} \qquad (result, fst \cdot \qquad evalSteps\ \$\ rest)
\end{array}
$$

An example

```
f <$> q = pSucceed f      <*>  q
p <*  q = const <$> p     <*>  q
f <$  q = const f         <$>  q

pInt    =      0     <$ pSym '0'
         <|> ...
         <|> 9       <$ pSym '9'
pPlus   =      (+)   <$> pInt <* pSym '+' <*> pInt
pTimes  =      (*)   <$> pInt <* pSym '*' <*> pInt

pExpr   = pPlus <|> pTimes
```

Parsing $1 + 2$

The result of parsing $1 + 2$ with $pPlus$ is:

$$
\begin{array}{llllll}
App \cdot & App \cdot & App & \cdot\ Val & const & \cdot \\
 & & & App \cdot & Val\ (+) & \cdot \\
 & & & & Shift & \cdot\ App \cdot\ Val\ (const\ 1) \cdot \\
 & & & & & \quad Val\ \texttt{'1'} \cdot \\
 & & Shift & \cdot\ Val\ \texttt{'+'} \cdot & & \\
 & Shift \cdot & App & \cdot\ Val\ (const\ 2) \cdot & & \\
 & & & Val\ \texttt{'2'} & & \\
Done & & & & &
\end{array}
$$

Parsing $1 + 2$

The result of parsing $1 + 2$ with $pPlus$ is:

$$
\begin{array}{lllll}
App \cdot & App \cdot & App \cdot & Val & const \cdot \\
 & & & App \cdot & Val\ (+) \cdot \\
 & & & & Shift \cdot App \cdot Val\ (const\ 1) \cdot \\
 & & & & \quad Val\ \texttt{'1'} \cdot \\
 & & Shift \cdot & Val\ \texttt{'+'} \cdot & \\
 & Shift \cdot & App \cdot & Val\ (const\ 2) \cdot & \\
 & & & Val\ \texttt{'2'} & \\
Done & & & &
\end{array}
$$

and with $pTimes$:

$$
\begin{array}{llll}
App \cdot App \cdot & App \cdot & Val & const \cdot \\
 & & App \cdot & Val\ (*) \cdot \\
 & & & Shift \cdot App \cdot Val\ (const\ 1) \cdot \\
 & & & \quad Val\ \texttt{'1'} \cdot \\
 & Fail & & &
\end{array}
$$

What happens to the function $best$

- $best$ thusfar deals with $Progress$, not with $Polish$

What happens to the function $best$

- $best$ thusfar deals with $Progress$, not with $Polish$
- but we may freely change the interleaving of $Progress$ and $Polish$ steps

What happens to the function $best$

- $best$ thusfar deals with $Progress$, not with $Polish$
- but we may freely change the interleaving of $Progress$ and $Polish$ steps
- provided we leave each of the subsequences intact

What happens to the function *best*

- *best* thusfar deals with *Progress*, not with *Polish*
- but we may freely change the interleaving of *Progress* and *Polish* steps
- provided we leave each of the subsequences intact
  getprogress moves progress information to the front of the sequence

$$
\begin{array}{lll}
\ldots & & \\
Shift\ v\ \text{`}best\text{`}\ Shift\ w & = & Shift\ (v\ \text{`}best\text{`}\ w) \\
p\ \ \text{`}best\text{`}\ q & = & getProgress\ id\ p \\
 & & \quad \text{`}best\text{`} \\
 & & getProgress\ id\ q
\end{array}
$$

$getProgress$

$$
\begin{array}{lll}
getProgress\ f\ (Val\ a\ \ s) & = getProgress & (f \cdot Val\ \ a\ )\ \ s \\
getProgress\ f\ (App\ \ \ \ s) & = getProgress & (f \cdot App\ \ \ \ )\ \ s \\
getProgress\ f\ (Done\ \ p) & = Done & (f\ \ p) \\
getProgress\ f\ (Shift\ \ s) & = Shift & (f\ \ s) \\
getProgress\ f\ (Fail\ ) & = Fail &
\end{array}
$$

$getProgress$

$$
\begin{array}{lll}
getProgress\ f\ (Val\ a\ \ s) & = getProgress\ & (f \cdot Val\ \ a\ )\ \ s \\
getProgress\ f\ (App\ \ \ \ s) & = getProgress\ & (f \cdot App\ \ \ \ )\ \ s \\
getProgress\ f\ (Done\ \ p) & = Done & (f\ \ p) \\
getProgress\ f\ (Shift\ \ s) & = Shift & (f\ \ s) \\
getProgress\ f\ (Fail\ ) & = Fail &
\end{array}
$$

does not work, since we call $getProgress$ on the "continuation type" s

$getProgress$

$$
\begin{array}{lll}
getProgress\ f\ (Val\ a\ \ s) & = getProgress\ (f \cdot Val\ \ a\ )\ \ s \\
getProgress\ f\ (App\ \ \ \ s) & = getProgress\ (f \cdot App\ \ \ \ )\ \ s \\
getProgress\ f\ (Done\ \ p) & = Done\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ (f\ \ p) \\
getProgress\ f\ (Shift\ \ s) & = Shift\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ (f\ \ s) \\
getProgress\ f\ (Fail\ ) & = Fail
\end{array}
$$

does not work, since we call $getProgress$ on the "continuation type" s
But the problem can be solved with some class hacking:

$$
\begin{array}{l}
\textbf{class}\ HasProgress\ st\ \textbf{where} \\
\quad getProgress :: (st \rightarrow Steps\ x\ y) \rightarrow st \rightarrow Steps\ x\ y \\
\\
\textbf{instance}\ HasProgress\ s \Rightarrow HasProgress\ (Steps\ a\ s)\ \textbf{where} \\
\quad \ldots \text{as before} \ldots \\
\\
\textbf{instance}\ HasProgress\ ()\ \textbf{where} \\
\quad getProgress\ f\ () = Fail
\end{array}
$$

Once more $1 + 2$

$$
\begin{array}{llllll}
App \cdot & App \cdot & App \cdot & Val & const & \cdot \\
 & & & App \cdot & Val\ (+) & \cdot \\
 & & & & Shift & \cdot App \cdot\ Val\ (const\ 1) \cdot \\
 & & & & & \quad Val\ \texttt{'1'} \cdot \\
 & & Shift \cdot & Val\ \texttt{'+'} \cdot & & \\
 & Shift \cdot & App \cdot & Val\ (const\ 2) \cdot & & \\
 & & & Val\ \texttt{'2'} & & \\
Done & & & & &
\end{array}
$$

$$
\begin{array}{llllll}
App \cdot & App \cdot & App \cdot & Val & const & \cdot \\
 & & & App \cdot & Val\ (*) & \cdot \\
 & & & & Shift & \cdot App \cdot\ Val\ (const\ 1) \cdot \\
 & & & & & \quad Val\ \texttt{'1'} \cdot \\
 & & Fail & & &
\end{array}
$$

Sequences after calling $getProgress$

$$
\begin{array}{l}
Shift\ \cdot \\
App \cdot\ App\ \cdot\ App\ \cdot\ Val\ \ const\ \cdot \\
\qquad\qquad\qquad App \cdot\ Val\ (+)\ \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad \cdot\ App \cdot\ Val\ (const\ 1) \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Val\ \texttt{'1'} \cdot \\
\qquad\qquad Shift\ \cdot\ Val\ \texttt{'+'} \cdot \\
\qquad Shift \cdot App\ \cdot\ Val\ (const\ 2) \cdot \\
\qquad\qquad\qquad\quad Val\ \texttt{'2'} \\
Done
\end{array}
$$

$$
\begin{array}{l}
Shift. \\
App \cdot App\ \cdot\ App\ \cdot\ Val\ \ const\ \cdot \\
\qquad\qquad\qquad App \cdot\ Val\ (*) \\
\qquad\qquad\qquad\qquad\qquad\qquad \cdot\ App \cdot\ Val\ (const\ 1) \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Val\ \texttt{'1'} \cdot \\
\qquad\qquad Fail
\end{array}
$$

Remove the $Shift$ steps

$$
\begin{array}{lllll}
App \cdot App \cdot App & \cdot\ Val & const & \cdot & \\
 & App \cdot & Val\ (+) & \cdot & \\
 & & & \cdot\ App \cdot & Val\ (const\ 1) \cdot \\
 & & & & Val\ \texttt{'1'} \cdot \\
Shift & \cdot\ Val\ \texttt{'+'} \cdot & & & \\
Shift \cdot App & \cdot\ Val\ (const\ 2) \cdot & & & \\
 & Val\ \texttt{'2'} & & & \\
Done & & & &
\end{array}
$$

$$
\begin{array}{lllll}
App \cdot App \cdot App & \cdot\ Val & const & \cdot & \\
 & App \cdot & Val\ (*) & & \\
 & & & \cdot\ App \cdot & Val\ (const\ 1) \cdot \\
 & & & & Val\ \texttt{'1'} \cdot \\
Fail & & & &
\end{array}
$$

... and $getProgress$ again ...

$$
\begin{array}{l}
Shift \\
App \cdot \; App \; \cdot App \; \cdot \; Val \quad const \quad \cdot \\
\qquad\qquad\qquad\qquad App \cdot \; Val \; (+) \; \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot App \cdot \; Val \; (const \; 1) \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Val \; \texttt{'1'} \cdot \\
\qquad\qquad\qquad\qquad Val \; \texttt{'+'} \cdot \\
\qquad\quad Shift \cdot App \; \cdot \; Val \; (const \; 2) \cdot \\
\qquad\qquad\qquad\qquad Val \; \texttt{'2'} \\
Done
\end{array}
$$

$$
\begin{array}{l}
Fail \\
App \cdot App \; \cdot App \; \cdot \; Val \quad const \; \cdot \\
\qquad\qquad\qquad\qquad App \cdot \; Val \; (*) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot App \cdot \; Val \; (const \; 1) \cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Val \; \texttt{'1'} \cdot
\end{array}
$$

... and *best* selects the first ...

$$
\begin{array}{l}
Shift \\
App \cdot App \cdot App \cdot Val\ const \cdot \\
\qquad App \cdot Val\ (+) \cdot \\
\qquad\qquad \cdot App \cdot Val\ (const\ 1) \cdot \\
\qquad\qquad\qquad Val\ \texttt{'1'} \cdot \\
\qquad Val\ \texttt{'+'} \cdot \\
\quad Shift \cdot App \cdot Val\ (const\ 2) \cdot \\
\qquad Val\ \texttt{'2'} \\
Done
\end{array}
$$

Monadic Interface: Example of its use

$$
\begin{aligned}
&\mathbf{data}\ XML\ = Tag\ t\ [XML] \\
&pMany\ p\quad = (:)\ \texttt{<\$>}\ p\ \texttt{<*>}\ pMany\ p\ \texttt{<|>}\ pSucceed\ [\,] \\
\\
&pXML = \mathbf{do}\ t \leftarrow pOpenTag \\
&\qquad\qquad Tag\ t\ \texttt{<\$>}\ pMany\ pXML\ \texttt{<*}\ pCloseTag\ t
\end{aligned}
$$

- we use it to parameterise a parser with a recognised result
- laziness may get lost if we use it to construct a result

Monadic interface I

Suppose we know the result a of the parser p that serves as the left operand in a monadic composition, then the result of a parser

$$P\ (\lambda k\ input \rightarrow p\ (q\ a\ k)\ input)$$

will be of type $Steps\ a\ (Steps\ b\ s)$.

Monadic interface I

Suppose we know the result a of the parser p that serves as the left operand in a monadic composition, then the result of a parser

$$P\ (\lambda k\ input \rightarrow p\ (q\ a\ k)\ input)$$

will be of type $Steps\ a\ (Steps\ b\ s)$.
We start by defining the counterpart of $getProgress$:

$$\begin{array}{lcl}
getVal :: Steps\ a\ (Steps\ b\ s) \rightarrow (a, Steps\ b\ s) & & \\
getVal\ (Val\ a\ \ s) & = & (a, s) \\
getVal\ (App\ \ \ \ s) & = & \mathbf{let}\ (a, s') \ = getVal\ s \\
 & & \quad\ \ (f, s'') \ = getVal\ s' \\
 & & \mathbf{in}\ \ (f\ a, s'') \\
getVal\ (Shift\ \ v) & = & \mathbf{let}\ (a, r) = \ getVal\ v\ \mathbf{in}\ (a, Shift\ \ r) \\
getVal\ (Done\ \ v) & = & \mathbf{let}\ (a, r) = \ getVal\ v\ \mathbf{in}\ (a, Done\ \ r) \\
getVal\ (Fail) & = & (\bot, Fail)
\end{array}$$

This function evaluates and removes the first value of from the sequence.

Monadic Interface II

```
instance Monad (Parser s) where
  return v   = pSucceed v
  P p >>= q  = P (λk input →
                    let (a, ps_qres) =
                          getVal (p (unP (q a) k) input)
                    in  ps_qres
                 )
```

Ambigous grammars

- remember we deal with alternatives in parallel
- so reduce-reduce stateshave to be identified
- we start producing results as soon as we know there is a single alternative left

Ambigous grammars

- remember we deal with alternatives in parallel
- so reduce-reduce stateshave to be identified
- we start producing results as soon as we know there is a single alternative left
- or the potential source of the ambiguity is gone

Ambigous grammars

- remember we deal with alternatives in parallel
- so reduce-reduce stateshave to be identified
- we start producing results as soon as we know there is a single alternative left
- or the potential source of the ambiguity is gone
- we have to annotate the grammar

Ambigous grammars

- remember we deal with alternatives in parallel
- so reduce-reduce stateshave to be identified
- we start producing results as soon as we know there is a single alternative left
- or the potential source of the ambiguity is gone
- we have to annotate the grammar

$$amb :: Par\ a \rightarrow Par\ [a]$$

Ambigous grammars

- remember we deal with alternatives in parallel
- so reduce-reduce stateshave to be identified
- we start producing results as soon as we know there is a single alternative left
- or the potential source of the ambiguity is gone
- we have to annotate the grammar

$$amb :: Par\ a \rightarrow Par\ [a]$$

- introduce a label to indicate where an ambigous non-terminal is reduced:

$$\begin{array}{lcl} \textbf{data}\ Steps\ a\ r & = & \ldots as\ before \ldots \\ & \mid & RS\ [Steps\ a\ r]\ (Steps\ a\ r) \end{array}$$

Extend the function *best*

$$
\begin{array}{ll}
best :: & HasProgress\ r \Rightarrow \\
& Steps\ a\ r \rightarrow Steps\ a\ r \rightarrow Steps\ a\ r \\
\ldots &
\end{array}
$$

$$
\begin{array}{llll}
RS\ as & ac\ `best`\ RS\ bs & bc & = RS\ (as +\!\!+ bs) \\
 & & & \quad (ac\ `best`\ bc) \\
RS\ as & ac\ `best`\ r@(Shift\ \_) & & = RS\ as\ (ac\ `best`\ r) \\
l@(Shift\ \_) & `best`\ RS\ bs & bc & = RS\ bs\ (l\ `best`\ bc)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ HasProgress\ s \Rightarrow HasProgress\ (Steps\ a\ s) \\
\quad \ldots \\
\quad getProgress\ f\ (RS\ r\ s) \quad = RS\ (map\ f\ r)\ (f\ s)
\end{array}
$$

The function amb

After adding the function $unRS$ that removes the RS marks, the function amb can be defined:

$$
\begin{aligned}
& amb\ \ (P\ p) \\
& \ = P\ \ (\lambda k\ input \\
& \qquad \rightarrow unRS\ id\ (p\ (\lambda inp \rightarrow RS\ [k\ inp]\ Fail)\ input) \\
& \qquad)
\end{aligned}
$$

$unRS$ behaves like $getProgress$ ...

$$
\begin{array}{lcl}
unRS\ f\ (Val\ a\ \ r) & = & unRS\ (f \cdot Val\ a)\ r \\
unRS\ f\ (App\ \ r) & = & unRS\ (f \cdot App\ \ )\ r \\
unRS\ f\ (Shift\ \ r) & = & Shift\ (unRS\ \ f\ \ r) \\
unRS\ f\ (Done\ \ r) & = & error\ \texttt{"incorrect program"} \\
unRS\ f\ Fail & = & Fail \\
unRS\ f\ (RS\ ls\ \ r) & & \\
\quad = \mathbf{let} & res = map\ (evalSteps \cdot f)\ ls & \\
\quad\ \ \mathbf{in} & Val\ (map\ fst\ res)\ (snd\ (head\ res)) & \\
 & \quad `best` & \\
 & unRS\ \ f\ \ r &
\end{array}
$$

$unRS$ behaves like $getProgress$ ...

$$
\begin{array}{lcl}
unRS\ f\ (Val\ a\ \ r) & = & unRS\ (f \cdot Val\ a)\ r \\
unRS\ f\ (App\ \ r) & = & unRS\ (f \cdot App\ \ )\ r \\
unRS\ f\ (Shift\ r) & = & Shift\ (unRS\ \ f\ \ r) \\
unRS\ f\ (Done\ r) & = & error\ \texttt{"incorrect program"} \\
unRS\ f\ Fail & = & Fail \\
unRS\ f\ (RS\ ls\ \ r) & & \\
\quad = \mathbf{let} & & res = map\ (evalSteps \cdot f)\ ls \\
\quad\ \ \mathbf{in} & & Val\ (map\ fst\ res)\ (snd\ (head\ res)) \\
 & & \quad `best` \\
 & & unRS\ \ f\ \ r
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{class}\ HasProgress\ st\ \mathbf{where}} \\
getProgress & :: & (st\ \rightarrow Steps\ x\ y) \rightarrow st \rightarrow Steps\ x\ \ \ y \\
unRS & :: & HasProgress\ y \Rightarrow \\
 & & (st\ \rightarrow Steps\ x\ y) \rightarrow st \rightarrow Steps\ [x]\ \ y
\end{array}
$$

Conclusions I

- the *Polish* transformation can be applied to every data type

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{RoseTree}\ a & = & \mathit{Node}\ a\ [\mathit{RoseTree}\ a] \\
 & \mid & \mathit{Leaf} \\
 & & \\
\textbf{type}\ \mathit{IntRoseTree} & = & \mathit{RoseTree}\ \mathit{Int}
\end{array}
$$

Conclusions I

- the *Polish* transformation can be applied to every data type

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{RoseTree}\ a & = & \mathit{Node}\ a\ [\mathit{RoseTree}\ a] \\
 & \mid & \mathit{Leaf} \\
 & & \\
\textbf{type}\ \mathit{IntRoseTree} & = & \mathit{RoseTree}\ \mathit{Int}
\end{array}
$$

becomes:

$$
\begin{array}{llcll}
\textbf{data}\ \mathit{RoseTree}'\ a'\ r & = & \mathit{Node}' & (a'\ (\mathit{List}\ (\mathit{Rosetree}'\ a)\ r)) & \\
 & \mid & \mathit{Leaf}' & r & \\
\textbf{data}\ \mathit{List}'\ f'\ r & = & \mathit{Cons}' & (f'\ (\mathit{List}'\ f'\ r)) & \\
 & \mid & \mathit{Nil}' & r & \\
\textbf{data}\ \mathit{Int}'\ r & = & \mathit{Int}' & \mathit{Int}\ r & \\
 & & & & \\
\textbf{type}\ \mathit{IntRoseTree}' & = & \mathit{RoseTree}'\ \mathit{Int}' & &
\end{array}
$$

Conclusions II

- this talk has been more about searching than parsing

Conclusions II

- this talk has been more about searching than parsing
  - build a search tree containing branching points and successful steps
  - make your result type "Polish"

Conclusions II

- this talk has been more about searching than parsing
  - build a search tree containing branching points and successful steps
  - make your result type "Polish"
  - you may include other subsequences, for reporting about:
    - what was actually expected at a point
    - what repair steps were taken

Conclusions II

- this talk has been more about searching than parsing
  - build a search tree containing branching points and successful steps
  - make your result type "Polish"
  - you may include other subsequences, for reporting about:
    - what was actually expected at a point
    - what repair steps were taken
  - you may associate penalties with steps