

# 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

**infixl** 3 <|>

**infixl** 4 <\*>, <\*, <\$, <\$>

**type** *Parser a* = *String* → [(*Expr a*, *String*)]

(<|>) :: *Parser a* → *Parser a* → *Parser a*

(<\*>) :: *Parser (b → a)* → *Parser b* → *Parser a*

*pSucceed* :: *a* → *Parser a*

*pFail* :: *Parser a*

*pSym* :: *Char* → *Parser a*

# The Type *Expr*

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

```
data Expr a =      Val a
               |  $\exists b.$  App (Expr (b  $\rightarrow$  a)) (Expr b)
```



# List of Successes Implementation: backtracking

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

# 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

<b>data</b> <i>Progress</i>	=	<i>Shift Progress</i>	— successful parsing step
		<i>Done</i>	— successful completion
		<i>Fail</i>	— no progress

# Recognisers: continuation style

```
newtype Rec = R (String → Progress)
                → (String → Progress)
rSucceed     = R (λ k input → k input)
rFail        = R (λ k input → Fail)
R p <*> R q   = R (λ k input → p (q k) input)
R p <|> R q   = R (λ k input → p k input
                    'best'
                    q k input )
pSym a       = R (λ k input →
                    case input of
                    (s : ss) → if a ≡ s
                        then Shift (k ss)
                        else Fail
                    []      → Fail
                    )
recognize (R r) input = r Done input
```

# Comparing Alternatives Depth-First

$l$  'best'  $r =$  **case** *last l* **of**  
    *Fail*  $\rightarrow r$   
    *Done*  $\rightarrow$  **case** *last r* **of**  
        *Fail*  $\rightarrow l$   
        *Done*  $\rightarrow$  *error* "ambiguous grammar"

*last (Shift l)* = *last l*  
*last Fail* = *Fail*  
*last Done* = *Done*

# Comparing Alternatives Depth-First

$l$  'best'  $r =$  **case**  $last\ l$  **of**  
     $Fail \rightarrow r$   
     $Done \rightarrow$  **case**  $last\ r$  **of**  
         $Fail \rightarrow l$   
         $Done \rightarrow error$  "ambiguous grammar"

$last\ (Shift\ l) = last\ l$   
 $last\ Fail = Fail$   
 $last\ Done = Done$

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

# Comparing Alternatives Breadth-First

<i>Fail</i>	'best' <i>p</i>	= <i>p</i>
<i>q</i>	'best' <i>Fail</i>	= <i>q</i>
<i>Done</i>	'best' <i>Done</i>	= error "ambiguous grammar"
<i>Done</i>	'best' <i>q</i>	= <i>Done</i>
<i>p</i>	'best' <i>Done</i>	= <i>Done</i>
<i>(Shift v)</i>	'best' <i>(Shift w)</i>	= <i>Shift (v 'best' w)</i>

# Comparing Alternatives Breadth-First

<i>Fail</i>	'best' <i>p</i>	= <i>p</i>
<i>q</i>	'best' <i>Fail</i>	= <i>q</i>
<i>Done</i>	'best' <i>Done</i>	= <i>error</i> "ambiguous grammar"
<i>Done</i>	'best' <i>q</i>	= <i>Done</i>
<i>p</i>	'best' <i>Done</i>	= <i>Done</i>
<i>(Shift v)</i>	'best' <i>(Shift w)</i>	= <i>Shift (v 'best' w)</i>

- ▶ the last alternative makes the comparison breadth-first
- ▶ the uses of *best* actually drive the parsing process, by asking recognisers to produce *Progress* 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 successfully 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

```
data Expr a = Val a
            | ∃b. App (Expr (b → a)) (Expr b)
```

# The Polish Data Type

- ▶ We extend the type with a continuation parameter, so

$$\begin{aligned} \text{data } \mathit{Expr} \ a &= \quad \mathit{Val} \ a \\ &| \ \exists b. \ \mathit{App} \ (\mathit{Expr} \ (b \rightarrow a)) \ (\mathit{Expr} \ b) \end{aligned}$$

- ▶ becomes

$$\begin{aligned} \text{data } \mathit{Polish} \ a \ s &= \quad \mathit{Val} \ a \ s \\ &| \ \exists b. \ \mathit{App} \ (\mathit{Polish} \ (b \rightarrow a)) \ (\mathit{Polish} \ b \ s) \end{aligned}$$



# 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:

```
data Steps a s =      Val      a s
                    |  ∃b. App   (Steps (b → a) (Steps b s))
                    |      Shift (Steps a s)
                    |      Done  (Steps a s)
                    |      Fail
```

# Interleaving Polish and Progress

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

```
data Steps a s =      Val    a s
                    |      ∃b. App    (Steps (b → a) (Steps b s))
                    |      Shift (Steps a s)
                    |      Done  (Steps a s)
                    |      Fail
```

- ▶ and the evaluator ignores *Progress* steps

```
evalSteps (Val a s) = (a, s)
evalSteps (App f s) = let (f, s') = evalSteps s
                        (a, s'') = evalSteps s'
                        in (f a, s'')
evalSteps (Shift v) = evalSteps v
evalSteps (Done v) = evalSteps v
evalSteps Fail    = error "wrong input"
```

# Parsers

```
pSucceed a           = P (Val a.)
P p    <*> P q      = P ((App . ) . p . q)
pSym    a
  = P (λ k input →
      case input of
        (s : ss) → if a ≡ s
                    then (Shift . Val s . k) ss
                    else Fail
        []       → Fail
      )
parse p input
  = let (result, rest) =
        evalSteps (p (λrest → Done (Val rest ())))
          input
      in (result, fst . evalSteps $ rest)
```

# An example

$f \langle \$ \rangle q = pSucceed\ f \quad \langle * \rangle \quad q$

$p \langle * \rangle q = const \langle \$ \rangle p \quad \langle * \rangle \quad q$

$f \langle \$ \rangle q = const\ f \quad \langle \$ \rangle \quad q$

$pInt = 0 \quad \langle \$ \rangle pSym\ '0'$

$\langle | \rangle \dots$

$\langle | \rangle 9 \quad \langle \$ \rangle pSym\ '9'$

$pPlus = (+) \quad \langle \$ \rangle pInt \langle * \rangle pSym\ '+' \langle * \rangle pInt$

$pTimes = (*) \quad \langle \$ \rangle pInt \langle * \rangle pSym\ '*' \langle * \rangle pInt$

$pExpr = pPlus \langle | \rangle pTimes$

# Parsing 1 + 2

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

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (+) ·  
*Shift* · *App* · *Val* (*const* 1) ·  
*Val* '1' ·  
*Shift* · *Val* '+' ·  
*Shift* · *App* · *Val* (*const* 2) ·  
*Val* '2' ·  
*Done*

# Parsing 1 + 2

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

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (+) ·  
*Shift* · *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Shift* · *Val* '+' ·  
*Shift* · *App* · *Val* (*const* 2) ·  
*Val* '2'

*Done*

and with *pTimes*:

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (\*) ·  
*Shift* · *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Fail*

# 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



# *getProgress*

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

# *getProgress*

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

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

# *getProgress*

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

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

**class** *HasProgress* *st* **where**

*getProgress* :: (*st* → *Steps* *x y*) → *st* → *Steps* *x y*

**instance** *HasProgress* *s* ⇒ *HasProgress* (*Steps* *a s*) **where**  
... as before...

**instance** *HasProgress* () **where**  
*getProgress* *f* () = *Fail*

# Once more 1 + 2

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (+) ·  
*Shift* · *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Shift* · *Val* '+' ·  
*Shift* · *App* · *Val* (*const* 2) ·  
*Val* '2'

*Done*

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (\*) ·  
*Shift* · *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Fail*

# Sequences after calling *getProgress*

*Shift* ·

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (+) ·  
· *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Shift* · *Val* '+' ·

*Shift* · *App* · *Val* (*const* 2) ·  
*Val* '2'

*Done*

*Shift*.

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (\*)  
· *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Fail*





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

*Shift*

*App* · *App* · *App* · *Val* *const* ·  
*App* · *Val* (+) ·  
· *App* · *Val* (*const* 1) ·  
*Val* '1' ·

*Val* '+' ·

*Shift* · *App* · *Val* (*const* 2) ·

*Val* '2' ·

*Done*

# Monadic Interface: Example of its use

```
data XML = Tag t [XML]
pMany p   = (:) <$> p <*> pMany p <|> pSucceed []

pXML = do t ← pOpenTag
        Tag t <$> pMany pXML <*> pCloseTag t
```

- ▶ 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

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

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

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

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

We start by defining the counterpart of  $\text{getProgress}$ :

$$\text{getVal} :: \text{Steps } a (\text{Steps } b s) \rightarrow (a, \text{Steps } b s)$$

$$\text{getVal } (\text{Val } a \ s) = (a, s)$$

$$\begin{aligned} \text{getVal } (\text{App } \ s) = & \text{let } (a, s') = \text{getVal } s \\ & (f, s'') = \text{getVal } s' \\ & \text{in } (f \ a, s'') \end{aligned}$$

$$\text{getVal } (\text{Shift } v) = \text{let } (a, r) = \text{getVal } v \text{ in } (a, \text{Shift } r)$$

$$\text{getVal } (\text{Done } v) = \text{let } (a, r) = \text{getVal } v \text{ in } (a, \text{Done } r)$$

$$\text{getVal } (\text{Fail}) = (\perp, \text{Fail})$$

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  $\gg$  q = P ( $\lambda k$  input  $\rightarrow$   
                let (a, ps_gres) =  
                    getVal (p (unP (q a) k) input)  
                in ps_gres  
                )
```

# Ambiguous grammars

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

# Ambiguous grammars

- ▶ remember we deal with alternatives in parallel
- ▶ so reduce-reduce states have 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

# Ambiguous grammars

- ▶ remember we deal with alternatives in parallel
- ▶ so reduce-reduce states have 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

# Ambiguous grammars

- ▶ remember we deal with alternatives in parallel
- ▶ so reduce-reduce states have 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

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

# Ambiguous grammars

- ▶ remember we deal with alternatives in parallel
- ▶ so reduce-reduce states have 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 ambiguous non-terminal is reduced:

```
data Steps a r = ... as before ...
                | RS [Steps a r] (Steps a r)
```

# Extend the function *best*

*best* :: *HasProgress* *r*  $\Rightarrow$   
*Steps* *a* *r*  $\rightarrow$  *Steps* *a* *r*  $\rightarrow$  *Steps* *a* *r*

...

*RS* *as* *ac* 'best' *RS* *bs* *bc* = *RS* (*as* ++ *bs*)  
(*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*)

**instance** *HasProgress* *s*  $\Rightarrow$  *HasProgress* (*Steps* *a* *s*)

...  
*getProgress* *f* (*RS* *r* *s*) = *RS* (*map* *f* *r*) (*f* *s*)

# The function *amb*

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

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

## *unRS* behaves like *getProgress* ...

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

# *unRS* behaves like *getProgress* ...

```
unRS f (Val a r)      =      unRS ( f · Val a ) r
unRS f (App  r)      =      unRS ( f · App  ) r
unRS f (Shift r)     = Shift ( unRS f          r )
unRS f (Done r)      = error "incorrect program"
unRS f Fail           = Fail
unRS f (RS ls r)
    = let res = map (evalSteps · f) ls
      in  Val (map fst res) (snd (head res))
          'best'
          unRS f r
```

```
class HasProgress st where
```

```
getProgress :: (st → Steps x y) → st → Steps x y
unRS        :: HasProgress y ⇒
                (st → Steps x y) → st → Steps [x] y
```

# Conclusions I

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

```
data RoseTree a = Node a [RoseTree a]  
                | Leaf
```

```
type IntRoseTree = RoseTree Int
```

# Conclusions I

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

```
data RoseTree a = Node a [RoseTree a]
                | Leaf
```

```
type IntRoseTree = RoseTree Int
```

becomes:

```
data RoseTree' a' r = Node' (a' (List (Rosetree' a) r))
                    | Leaf' r
```

```
data List' f' r = Cons' (f' (List' f' r))
                 | Nil' r
```

```
data Int' r = Int' Int r
```

```
type IntRoseTree' = RoseTree' Int'
```

# 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