Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Parser Memoisation

Marcelo Barbosa de Sousa

January 24, 2011

Roadmap

Roadmap

Motivation

YAPL - Yet Another Parsing Library

Memoisation

Conclusion

Research Problem

Currently all parsing libraries will parse a piece of text twice with the same non-terminal if the grammar has various way of reaching the same point in the input. One approach to solving this problem is to use **memoisation** *techniques. It is however not entirely clear how to do this in a* **online**, **typeful** *way.*

Research Problem

Currently all parsing libraries will parse a piece of text twice with the same non-terminal if the grammar has various way of reaching the same point in the input. One approach to solving this problem is to use **memoisation** *techniques. It is however not entirely clear how to do this in a* **online**, **typeful** *way.*

Parser Combinators

Context free grammars

$$< N, \Sigma, P, S >$$

- N : Set of non-terminals
- Σ : Set of terminals, e.g. *Char*
- P : Set of productions (A $\in$ N, $\alpha \in V^*$)
- S : Set of start symbols, S $\in$ N
- $V = N \cup \Sigma$: Set of symbols
- $N \cap \Sigma = \varnothing$

Parser Type

$$
\begin{array}{l}
\textbf{type}\ \mathit{Parser}\ s\ a = [s] \quad\ \ \rightarrow a \\
\textbf{type}\ \mathit{Parser}\ s\ a = [s] \quad\ \ \rightarrow (a, [s]) \\
\textbf{type}\ \mathit{Parser}\ s\ a = [s] \quad\ \ \rightarrow [a] \\
\textbf{type}\ \mathit{Parser}\ s\ a = [s] \quad\ \ \rightarrow [(a, [s])] \\
\textbf{type}\ \mathit{Parser}\ \ \ a = \mathit{String} \rightarrow [(a, \mathit{String})]
\end{array}
$$

Basic Parser Combinators Types

$$\begin{array}{lll} pSym & :: Eq\ s \Rightarrow s & \rightarrow Parser\ s\ s \\ pRet & :: a & \rightarrow Parser\ s\ a \\ <\mid> & :: Parser\ s\ a \rightarrow Parser\ s\ a & \rightarrow Parser\ s\ a \\ <*> & :: Parser\ s\ (b \rightarrow a) \rightarrow Parser\ s\ b & \rightarrow Parser\ s\ a \end{array}$$

Basic Parser Combinators

$$
\begin{array}{l}
pSym :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s \\
pSym\ c = \lambda inp \rightarrow \mathbf{case}\ inp\ \mathbf{of} \\
\quad (s : ss) \mid c \equiv s \rightarrow [(s, ss)] \\
\quad otherwise \rightarrow [\,]
\end{array}
$$

$$
\begin{array}{l}
pRet :: a \rightarrow Parser\ s\ a \\
pRet\ x = \lambda inp \rightarrow [(x, inp)]
\end{array}
$$

Basic Parser Combinators

$$
\begin{aligned}
&pSym :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s \\
&pSym\ c = \lambda inp \rightarrow \mathbf{case}\ inp\ \mathbf{of} \\
&\quad (s : ss) \mid c \equiv s \rightarrow [(s, ss)] \\
&\quad otherwise \rightarrow [\,]
\end{aligned}
$$

$$
\begin{aligned}
&pRet :: a \rightarrow Parser\ s\ a \\
&pRet\ x = \lambda inp \rightarrow [(x, inp)]
\end{aligned}
$$

Basic Parser Combinators

$$
\begin{aligned}
&\textbf{infixr}\ 3 <|> \\
&(<|>) :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
&p <|> q = \lambda \mathit{inp} \to p\ \mathit{inp} +\!\!+\ q\ \mathit{inp}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{infixl}\ 5 <*> \\
&(<*>) :: \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a \\
&p <*> q = \lambda \mathit{inp} \to [(\mathit{b2a}\ b, \mathit{rr}) \mid (\mathit{b2a}, r) \leftarrow p\ \mathit{inp}, \\
&\quad (b, \mathit{rr}) \leftarrow q\ r]
\end{aligned}
$$

Basic Parser Combinators

$$
\begin{aligned}
&\textbf{infixr}\ 3 <|>\\
&(<|>) :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a\\
&p <|> q = \lambda inp \to p\ inp +\!\!+\ q\ inp
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{infixl}\ 5 <*>\\
&(<*>) :: \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a\\
&p <*> q = \lambda inp \to [(b2a\ b, rr) \mid (b2a, r) \leftarrow p\ inp,\\
&\quad (b, rr) \leftarrow q\ r]
\end{aligned}
$$

Derived Parser Combinators

$$\textbf{infix}\ 7 <\$>$$
$$(<\$>) :: (b \to a) \to Parser\ s\ b \to Parser\ s\ a$$
$$f <\$> q = pRet\ f <*> q$$

$$\textbf{infixl}\ 3\ `opt`$$
$$opt :: Parser\ s\ a \to a \to Parser\ s\ a$$
$$p\ `opt`\ v = p <|> pReturn\ v$$

$$pMany :: Parser\ s\ a \to Parser\ s\ [a]$$
$$pMany\ p = (:) <\$> p <*> pMany\ p\ `opt`\ [\,]$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

The Research Problem

Grammars are not always left factorized which results in inefficient parsing.

$$s :: Parser\ Char\ String$$
$$s = f <|> g$$

$$f = fsem <\$> pSym\ \text{'a'} <*> p <*> q$$

$$g = h <*> p <*> r$$
$$h = gsem <\$> pSym\ \text{'a'}$$

The Research Problem

Grammars are not always left factorized which results in inefficient parsing.

$$s :: Parser\ Char\ String$$
$$s = f <|> g$$

$$f = fsem <\$> pSym\ \texttt{'a'} <*> p <*> q$$

$$g = h <*> p <*> r$$
$$h = gsem <\$> pSym\ \texttt{'a'}$$

The Research Problem

Grammars are not always left factorized which results in inefficient parsing.

$$s :: Parser\ Char\ String$$
$$s = f <|> g$$

$$f = fsem <\$> pSym\ \texttt{'a'} <*> p <*> q$$

$$g = h <*> p <*> r$$
$$h = gsem <\$> pSym\ \texttt{'a'}$$

The Research Problem

$$s :: Parser\ Char\ String$$
$$s = f < | > g$$

$$f = fsem < \$ > pSym\ \texttt{'a'} < * > p < * > q$$

$$g = h < * > p < * > r$$
$$h = gsem < \$ > pSym\ \texttt{'a'}$$

The inefficiency results of our implementation of the choice combinator.
In the case of parser s, $pSym\ 'a'\ < * >\ p$ could be shared among both branches.

Roadmap

Motivation

YAPL - Yet Another Parsing Library

Memoisation

Conclusion

Goals

- Left-factorization for free
- Full control on parsing, online parsing, error control
- Same representation as previous parser combinators

Defunctionalizing Parsers

We can represent parsers with GADTs:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Parser} :: * \to * \textbf{where}} \\
\quad \mathit{Sym} & :: \mathit{Char} & \to \mathit{Parser}\ \mathit{Char} \\
\quad \mathit{Ret} & :: a & \to \mathit{Parser}\ a \\
\quad \mathit{Alt} & :: \mathit{Parser}\ a \to \mathit{Parser}\ a & \to \mathit{Parser}\ a \\
\quad \mathit{Seq} & :: \mathit{Parser}\ (b \to a) \to \mathit{Parser}\ b & \to \mathit{Parser}\ a
\end{array}
$$

Continuations as a Stack of Parsers

We can represent the parsers yet to be analyzed as a stack.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Pending} :: * \rightarrow * \textbf{where} \\
\quad \mathit{Done} :: \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Pending}\ () \\
\quad \mathit{Stack} :: \mathit{Parser}\ a \rightarrow \mathit{Pending}\ b \rightarrow \mathit{Pending}\ (a, b)
\end{array}
$$

- *Done* represents the end of our parser.
- *Stack* contains the current parser to be processed and the rest of the stack.

```
   Ret (:[]) `Seq` Sym 'a'
⇝ Stack (Ret (:[])) (Stack (Sym 'a') Done)
:: Pending (a → [a], (Char, ()))
```

Continuations as a Stack of Parsers

We can represent the parsers yet to be analyzed as a stack.

$$\begin{array}{l} \textbf{data}\ \mathit{Pending} :: * \to * \textbf{where} \\ \quad \mathit{Done} :: \qquad\qquad\qquad\qquad\qquad\qquad \mathit{Pending}\ () \\ \quad \mathit{Stack} :: \mathit{Parser}\ a \to \mathit{Pending}\ b \to \mathit{Pending}\ (a,b) \end{array}$$

- *Done* represents the end of our parser.
- *Stack* contains the current parser to be processed and the rest of the stack.

```
   Ret (:[]) ‘Seq‘ Sym ’a’
⇝ Stack (Ret (:[])) (Stack (Sym ’a’) Done)
:: Pending (a → [a], (Char, ()))
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Continuations as a Stack of Parsers

We can represent the parsers yet to be analyzed as a stack.

$$\begin{aligned}
&\textbf{data}\ \mathit{Pending} :: * \to * \textbf{where}\\
&\quad \mathit{Done} :: \qquad\qquad\qquad\qquad\qquad\quad\ \mathit{Pending}\ ()\\
&\quad \mathit{Stack} :: \mathit{Parser}\ a \to \mathit{Pending}\ b \to \mathit{Pending}\ (a, b)
\end{aligned}$$

- *Done* represents the end of our parser.
- *Stack* contains the current parser to be processed and the rest of the stack.

```
   Ret (:[]) ‘Seq‘ Sym ’a’
⇝ Stack (Ret (:[])) (Stack (Sym ’a’) Done)
:: Pending (a → [a], (Char, ()))
```

Parsing as a State Machine

$$\textbf{data}\ \mathit{State}\ a = \forall b \circ \mathit{State}\ (b \to a)\ (\mathit{Pending}\ b)$$
$$\textbf{type}\ \mathit{States}\ a = [\mathit{State}\ a]$$

The use of existential types in *State* is crucial as we will see next.
Our interface function *runParser* simply builds the first state given a parser.

$$\mathit{runParser} :: \mathit{Parser}\ a \to [\mathit{Char}] \to [a]$$
$$\mathit{runParser}\ p = \mathit{parse}\ [\mathit{State}\ \mathit{fst}\ (\mathit{Stack}\ p\ \mathit{Done})]$$

Parsing as a State Machine

We can define the *parse* function *iteratively*:

$$
\begin{aligned}
&\mathit{parse} :: \mathit{States}\ a \rightarrow [\mathit{Char}] \rightarrow [a]\\
&\mathit{parse}\ \mathit{states}\ [\,] \qquad\quad = \mathit{evalStates}\ \mathit{states}\\
&\mathit{parse}\ \mathit{states}\ (x : \mathit{xs}) = (\mathit{transition}\ \mathit{states}\ x)\ \text{`}\mathit{parse}\text{`}\ \mathit{xs}
\end{aligned}
$$

For simplicity, I've consider [a] as the result of *evalStates*

$$
\begin{aligned}
&\mathit{evalStates} :: \mathit{States}\ a \rightarrow [a]\\
&\mathit{evalStates}\ [\,] \qquad\qquad\qquad\quad\;\; = [\,]\\
&\mathit{evalStates}\ ((\mathit{State}\ f\ \mathit{Done}) : \mathit{rest}) = (f\ ()) : \mathit{evalStates}\ \mathit{rest}\\
&\mathit{evalStates}\ (\_ : \mathit{rest}) \qquad\qquad\;\; = \mathit{error}\ \texttt{"parse error"}
\end{aligned}
$$

The parsing result type is flexible, and in *evalStates* we can do some error reporting although limited.

Parsing as a State Machine

We can define the *parse* function *iteratively*:

$$
\begin{aligned}
&parse :: States\ a \rightarrow [Char] \rightarrow [a]\\
&parse\ states\ [\,] \qquad\quad = evalStates\ states\\
&parse\ states\ (x : xs) = (transition\ states\ x)\ `parse`\ xs
\end{aligned}
$$

For simplicity, I've consider [a] as the result of *evalStates*

$$
\begin{aligned}
&evalStates :: States\ a \rightarrow [a]\\
&evalStates\ [\,] &&= [\,]\\
&evalStates\ ((State\ f\ Done) : rest) &&= (f\ ()) : evalStates\ rest\\
&evalStates\ (\_ : rest) &&= error\ \texttt{"parse error"}
\end{aligned}
$$

The parsing result type is flexible, and in *evalStates* we can do some error reporting although limited.

Parsing as a State Machine

$$
\begin{array}{l}
\mathit{transition} :: \mathit{States}\ a \rightarrow \mathit{Char} \rightarrow \mathit{States}\ a \\
\mathit{transition}\ \mathit{states}\ \mathit{char} = \\
\quad \mathbf{let}\ \mathit{unFoldNtStates} = \mathit{concatMap}\ \mathit{unFoldNtAtHeads}\ \mathit{states} \\
\quad \mathbf{in}\ \mathit{foldl}\ (\mathit{reduce}\ \mathit{char})\ [\,]\ \mathit{unFoldNtStates}
\end{array}
$$

In each *transition* we unfold non-terminals, which are parsers and put them on the stack until we have on top of the stack a parser that consumes the current input position.
For simplicity we consider that the input is a String.

Reducing Terminals

$$
\begin{aligned}
&\mathit{reduce} :: \mathit{Char} \to \mathit{States}\ a \to \mathit{State}\ a \to \mathit{States}\ a\\
&\mathit{reduce}\ \mathit{char}\ \mathit{states}\ (\mathit{State}\ f\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest}))\\
&\quad \mid c \equiv \mathit{char} \ \ = (\mathit{State}\ (\lambda \mathit{rest} \to f\ (\mathit{char}, \mathit{rest}))\ \mathit{rest}) : \mathit{states}\\
&\quad \mid \mathit{otherwise} = \mathit{states}\\
&\mathit{reduce}\ \_\ \mathit{states}\ s@(\mathit{State}\ f\ \mathit{Done}) = s : \mathit{states}\\
&\mathit{reduce}\ \_\ \mathit{states}\ \_ \qquad\qquad\qquad\quad\ = \mathit{states}
\end{aligned}
$$

At this point we are consuming the input character and generating a new list of states. We can introduce error reporting as well.

Unfolding Non Terminals

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads} :: \mathit{State}\ a \rightarrow \mathit{States}\ a\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ \mathit{Done}) && = [s]\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest})) && = [s]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda \mathit{rest} \rightarrow r\ (f, \mathit{rest}))\ \mathit{rest})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda(\mathit{pr}, (\mathit{qr}, \mathit{rest})) \rightarrow r\ ((\mathit{pr}\ \mathit{qr}), \mathit{rest}))\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Stack}\ q\ \mathit{rest})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Alt}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathbf{let}\ \mathit{statesp} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ p\ \mathit{rest})\\
&\qquad\quad \mathit{statesq} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ q\ \mathit{rest})\\
&\quad \mathbf{in}\ \ \mathit{statesp} + \! + \ \mathit{statesq}
\end{aligned}
$$

Unfolding Non Terminals

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads} :: \mathit{State}\ a \to \mathit{States}\ a\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ \mathit{Done}) &&= [s]\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest})) &&= [s]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda \mathit{rest} \to r\ (f, \mathit{rest}))\ \mathit{rest})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda(\mathit{pr}, (\mathit{qr}, \mathit{rest})) \to r\ ((\mathit{pr}\ \mathit{qr}), \mathit{rest}))\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Stack}\ q\ \mathit{rest})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Alt}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathbf{let}\ \mathit{statesp} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ p\ \mathit{rest})\\
&\qquad\ \ \ \mathit{statesq} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ q\ \mathit{rest})\\
&\quad \mathbf{in}\ \ \mathit{statesp} +\!\!+\ \mathit{statesq}
\end{aligned}
$$

Unfolding Non Terminals

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads} :: \mathit{State}\ a \to \mathit{States}\ a\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ \mathit{Done}) &&= [s]\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest})) &&= [s]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda \mathit{rest} \to r\ (f, \mathit{rest}))\ \mathit{rest})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda(\mathit{pr}, (\mathit{qr}, \mathit{rest})) \to r\ ((\mathit{pr}\ \mathit{qr}), \mathit{rest}))\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Stack}\ q\ \mathit{rest})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Alt}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathbf{let}\ \mathit{statesp} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ p\ \mathit{rest})\\
&\qquad\quad\ \mathit{statesq} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ q\ \mathit{rest})\\
&\quad \mathbf{in}\ \ \mathit{statesp} +\!\!+\ \mathit{statesq}
\end{aligned}
$$

Unfolding Non Terminals

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads} :: \mathit{State}\ a \rightarrow \mathit{States}\ a\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ \mathit{Done}) && = [s]\\
&\mathit{unFoldNtAtHeads}\ s@(\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest})) && = [s]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda \mathit{rest} \rightarrow r\ (f, \mathit{rest}))\ \mathit{rest})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathit{unFoldNtAtHeads}\ (\mathit{State}\ (\lambda(\mathit{pr}, (\mathit{qr}, \mathit{rest})) \rightarrow r\ ((\mathit{pr}\ \mathit{qr}), \mathit{rest}))\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Stack}\ q\ \mathit{rest})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Alt}\ p\ q)\ \mathit{rest})) =\\
&\quad \mathbf{let}\ \mathit{statesp} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ p\ \mathit{rest})\\
&\qquad\quad \mathit{statesq} = \mathit{unFoldNtAtHeads}\ \$\ \mathit{State}\ r\ (\mathit{Stack}\ q\ \mathit{rest})\\
&\quad \mathbf{in}\ \ \mathit{statesp} +\!\!+\ \mathit{statesq}
\end{aligned}
$$

Running example

$$pChar :: Char \to Parser\ String$$
$$pChar\ c = Ret\ (:[\,])\ `Seq`\ Sym\ c$$

```
> runParser (pChar 'a') "a"
["a"]
```

```
> runParser (pChar 'a') "ab"
["a"]
```

```
> runParser (pChar 'a' `Alt` pChar 'b') "a"
["a"]
> runParser (pChar 'a' `Alt` pChar 'b') "b"
["b"]
```

Running example

$$pChar :: Char \rightarrow Parser\ String$$
$$pChar\ c = Ret\ (:[\,])\ `Seq`\ Sym\ c$$

```
> runParser (pChar 'a') "a"
["a"]
```

```
> runParser (pChar 'a') "ab"
["a"]
```

```
> runParser (pChar 'a' `Alt` pChar 'b') "a"
["a"]
> runParser (pChar 'a' `Alt` pChar 'b') "b"
["b"]
```

Running example

$$
\begin{aligned}
&pChar :: Char \to Parser\ String \\
&pChar\ c = Ret\ (:[\,])\ `Seq`\ Sym\ c
\end{aligned}
$$

```
> runParser (pChar 'a') "a"
["a"]
```

```
> runParser (pChar 'a') "ab"
["a"]
```

```
> runParser (pChar 'a' `Alt` pChar 'b') "a"
["a"]
> runParser (pChar 'a' `Alt` pChar 'b') "b"
["b"]
```

Running example

$$pChar :: Char \rightarrow Parser\ String$$
$$pChar\ c = Ret\ (:[\,])\ \text{`}Seq\text{`}\ Sym\ c$$

```
> runParser (pChar 'a') "a"
["a"]
```

```
> runParser (pChar 'a') "ab"
["a"]
```

```
> runParser (pChar 'a' `Alt` pChar 'b') "a"
["a"]
> runParser (pChar 'a' `Alt` pChar 'b') "b"
["b"]
```

Running example

```
> runParser (pChar 'a') "b"
[]
```

```
> runParser (pChar 'a') ""
*** Exception: parse error
```

```
> runParser (Ret "a") ""
*** Exception: parse error
```

Running example

```
> runParser (pChar 'a') "b"
[]
```

```
> runParser (pChar 'a') ""
*** Exception: parse error
```

```
> runParser (Ret "a") ""
*** Exception: parse error
```

Running example

```
> runParser (pChar 'a') "b"
[]
```

```
> runParser (pChar 'a') ""
*** Exception: parse error
```

```
> runParser (Ret "a") ""
*** Exception: parse error
```

The Ret Problem

Since the Ret combinator does not require any input to succeed we need an extra step on evaluation:

$$
\begin{aligned}
&\mathit{evalStates}\ ((\mathit{State}\ f\ (\mathit{Stack}\ (\mathit{Ret}\ x)\ \mathit{Done})) : \mathit{rest}) = \\
&\quad (f\ (x, ())) : \mathit{evalStates}\ \mathit{rest}
\end{aligned}
$$

```
> runParser (Ret "a") ""
["a"]
```

```
> runParser (Alt (Ret "b") (Ret "a")) ""
["b","a"]
```

The Ret Problem

Since the Ret combinator does not require any input to succeed we need an extra step on evaluation:

$$
\begin{aligned}
&\mathit{evalStates}\ ((\mathit{State}\ f\ (\mathit{Stack}\ (\mathit{Ret}\ x)\ \mathit{Done})) : \mathit{rest}) = \\
&\quad (f\ (x, ())) : \mathit{evalStates}\ \mathit{rest}
\end{aligned}
$$

```
> runParser (Ret "a") ""
["a"]
```

```
> runParser (Alt (Ret "b") (Ret "a")) ""
["b","a"]
```

Simple extensions

It's easy to extend the library with other simple combinators such as pSatisfay.
We need to extend our *Parser* GADT

$$Sat :: (Char \to Bool) \to Parser\ Char$$

Since *Sat* consumes input we do not unfold anything

$$unFoldNtAtHeads\ s@(State\ r\ (Stack\ (Sat\ f)\ rest)) = [s]$$

Finally we need to create a new state in the reduce step

$$\begin{aligned} & reduce\ char\ states\ (State\ f\ (Stack\ (Sat\ s)\ rest)) \\ & \quad \mid s\ char \qquad = (State\ (\lambda rest \to f\ (char, rest))\ rest) : states \\ & \quad \mid otherwise = states \end{aligned}$$

Simple extensions

It's easy to extend the library with other simple combinators such as pSatisfay.
We need to extend our *Parser* GADT

$$Sat :: (Char \rightarrow Bool) \rightarrow Parser\ Char$$

Since *Sat* consumes input we do not unfold anything

$$unFoldNtAtHeads\ s@(State\ r\ (Stack\ (Sat\ f)\ rest)) = [s]$$

Finally we need to create a new state in the reduce step

$$\begin{aligned} &reduce\ char\ states\ (State\ f\ (Stack\ (Sat\ s)\ rest)) \\ &\quad \mid s\ char \quad\ \ = (State\ (\lambda rest \rightarrow f\ (char, rest))\ rest) : states \\ &\quad \mid otherwise = states \end{aligned}$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Simple extensions

It's easy to extend the library with other simple combinators such as pSatisfay.
We need to extend our *Parser* GADT

$$Sat :: (Char \to Bool) \to Parser\ Char$$

Since *Sat* consumes input we do not unfold anything

$$unFoldNtAtHeads\ s@(State\ r\ (Stack\ (Sat\ f)\ rest)) = [s]$$

Finally we need to create a new state in the reduce step

$$\begin{aligned} &reduce\ char\ states\ (State\ f\ (Stack\ (Sat\ s)\ rest)) \\ &\quad \mid s\ char \quad\ \ = (State\ (\lambda rest \to f\ (char, rest))\ rest) : states \\ &\quad \mid otherwise = states \end{aligned}$$

Simple extensions

$$
\begin{aligned}
&\mathit{pDigit} :: \mathit{Parser}\ \mathit{Char}\\
&\mathit{pDigit} = \mathit{Sat}\ (\lambda x \to \texttt{'0'} \leqslant x \wedge x \leqslant \texttt{'9'})\\
&\mathit{pDigitAsInt} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{pDigitAsInt}\ \ =\ \ \mathit{Ret}\ (\lambda c \to \mathit{fromEnum}\ c - \mathit{fromEnum}\ \texttt{'0'})\\
&\qquad\qquad\quad \text{`}\mathit{Seq}\text{`}\ \mathit{pDigit}
\end{aligned}
$$

```
> runParser pDigitAsInt "1"
[1]
```

```
> runParser pDigitAsInt "92"
[9]
```


Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Simple extensions

$$
\begin{aligned}
&\mathit{pDigit} :: \mathit{Parser}\ \mathit{Char}\\
&\mathit{pDigit} = \mathit{Sat}\ (\lambda x \to \texttt{'0'} \leqslant x \land x \leqslant \texttt{'9'})\\
&\mathit{pDigitAsInt} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{pDigitAsInt} \quad = \quad \mathit{Ret}\ (\lambda c \to \mathit{fromEnum}\ c - \mathit{fromEnum}\ \texttt{'0'})\\
&\qquad\qquad\qquad \text{`}\mathit{Seq}\text{`}\ \mathit{pDigit}
\end{aligned}
$$

```
> runParser pDigitAsInt "1"
[1]
```

```
> runParser pDigitAsInt "92"
[9]
```

Functionalizing again

We can represent our initial parsers combinators with our new library

$$
\begin{aligned}
&pReturn :: a \to Parser\ a \\
&pReturn = Ret \\
&pSym :: Char \to Parser\ Char \\
&pSym = Sym
\end{aligned}
$$

$$
\begin{aligned}
&(<|>) :: Parser\ a \to Parser\ a \to Parser\ a \\
&p <|> q = p\ `Alt`\ q
\end{aligned}
$$

$$
\begin{aligned}
&(<*>) :: Parser\ (b \to a) \to Parser\ b \to Parser\ a \\
&p <*> q = p\ `Seq`\ q
\end{aligned}
$$

Functionalizing again

We can represent our initial parsers combinators with our new library

$$pReturn :: a \to Parser\ a$$
$$pReturn = Ret$$
$$pSym :: Char \to Parser\ Char$$
$$pSym = Sym$$

$$(<|>) :: Parser\ a \to Parser\ a \to Parser\ a$$
$$p <|> q = p\ `Alt`\ q$$

$$(<*>) :: Parser\ (b \to a) \to Parser\ b \to Parser\ a$$
$$p <*> q = p\ `Seq`\ q$$

Functionalizing again

We can represent our initial parsers combinators with our new library

$$pReturn :: a \to Parser\ a$$
$$pReturn = Ret$$
$$pSym :: Char \to Parser\ Char$$
$$pSym = Sym$$

$$(<|>) :: Parser\ a \to Parser\ a \to Parser\ a$$
$$p <|> q = p\ `Alt`\ q$$

$$(<*>) :: Parser\ (b \to a) \to Parser\ b \to Parser\ a$$
$$p <*> q = p\ `Seq`\ q$$

Functionalizing again

$$pManyA :: Parser\ String$$
$$pManyA = pMany\ (pSym\ \texttt{'a'})$$

$$pManyAb :: Parser\ String$$
$$pManyAb = (\lambda as\ b \rightarrow as +\!\!+ [b])$$
$$<\$> pManyA <*> pSym\ \texttt{'b'}$$

```
> runParser pManyA ""
[""]
```

```
> runParser pManyA "a"
["","a"]
```

Functionalizing again

$$
\begin{aligned}
&pManyA :: Parser\ String\\
&pManyA = pMany\ (pSym\ \texttt{'a'})\\
&pManyAb :: Parser\ String\\
&pManyAb \quad = \quad (\lambda as\ b \to as +\!\!+ [b])\\
&\qquad\qquad < \$ > pManyA < * > pSym\ \texttt{'b'}
\end{aligned}
$$

```
> runParser pManyA ""
[""]
```

```
> runParser pManyA "a"
["","a"]
```

Functionalizing again

```
> runParser pMany A "aaaa"
["a","aaa","aaaa","aa",""]
```

```
> runParser pManyAb "aaaa"
*** Exception: parse error
```

```
> runParser pManyAb "aaaab"
["aaaab"]
```

State of art of our library

- suited for left-factorization
- easily extensible
- online parsing
- easy implementable control error
- easy to use

Roadmap

The Equality Problem

Equality on Parsers

Since we have a clear representation of parsers we define equality parsers in a natural way.

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ (Parser\ a)\ \textbf{where} \\
\quad \begin{array}{lllll}
Sym\ c & \equiv Sym\ d & = c \equiv d \\
Ret\ a & \equiv Ret\ b & = a \equiv b \\
Alt\ p\ q & \equiv Alt\ r\ s & = p \equiv r \wedge q \equiv s \\
Seq\ f\ p & \equiv Seq\ g\ q & = f \equiv g \wedge p \equiv q
\end{array}
\end{array}
$$

Unfortunately this won't type check, because in our *Parser* representation we loose the type of the type variable *b* in *Seq*.

Recursive descent parsing

Even if it type checked the following evaluation would not terminate.

```
> pManyA == pManyA
```

$$pManyA :: Parser\ String$$
$$pManyA = pMany\ (pSym\ \texttt{'a'})$$

Having recursive parsers difficulties equality.

Stable Pointers

A stable pointer is a reference to a Haskell expression that is guaranteed not to be affected by garbage collection, i.e., it will neither be deallocated nor will the value of the stable pointer itself change during garbage collection (ordinary references may be relocated during garbage collection). Consequently, stable pointers can be passed to foreign code, which can treat it as an opaque reference to a Haskell value.

Equality on Parsers with Stable Pointers

There is a predefined Eq instance for (StablePtr a) which helps with the equality of stable pointers, but unfortunately stable pointers live in the IO world. Furthermore the comparison of parsers in different states requires us to help the compiler to recognize that two parsers have the same type.

```haskell
data Equal :: * -> * -> * where
  Eq :: Equal a a
```

```haskell
eq :: Parser a -> Parser b -> Maybe (Equal a b)
eq p q = let pptr = unsafePerformIO $ castParser p
             qptr = unsafePerformIO $ newStablePtr q
         in  if pptr == qptr
               then Just $ unsafeCoerce Eq
               else Nothing
```

Equality on Parsers with Stable Pointers

There is a predefined Eq instance for (StablePtr a) which helps with the equality of stable pointers, but unfortunately stable pointers live in the IO world. Furthermore the comparison of parsers in different states requires us to help the compiler to recognize that two parsers have the same type.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Equal} :: * \to * \to * \textbf{where} \\
\quad \mathit{Eq} :: \mathit{Equal}\ a\ a
\end{array}
$$

$$
\begin{array}{l}
\mathit{eq} :: \mathit{Parser}\ a \to \mathit{Parser}\ b \to \mathit{Maybe}\ (\mathit{Equal}\ a\ b) \\
\mathit{eq}\ p\ q = \textbf{let}\ \mathit{pptr} = \mathit{unsafePerformIO}\ \$\ \mathit{castParser}\ p \\
\qquad\qquad\qquad \mathit{qptr} = \mathit{unsafePerformIO}\ \$\ \mathit{newStablePtr}\ q \\
\qquad\quad\ \ \textbf{in}\ \ \textbf{if}\ \mathit{pptr} \equiv \mathit{qptr} \\
\qquad\qquad\qquad\ \ \textbf{then}\ \mathit{Just}\ \$\ \mathit{unsafeCoerce}\ \mathit{Eq} \\
\qquad\qquad\qquad\ \ \textbf{else}\ \mathit{Nothing}
\end{array}
$$

Equality on Parsers with Stable Pointers

$$
\begin{array}{l}
\mathit{castParser} :: \mathit{Parser}\ a \rightarrow \mathit{IO}\ (\mathit{StablePtr}\ (\mathit{Parser}\ b))\\
\mathit{castParser}\ p =\\
\quad \mathbf{do}\ \mathit{sptr} \leftarrow \mathit{newStablePtr}\ p\\
\qquad\ \ \mathit{return}\ \$\ \mathit{castPtrToStablePtr}\ \$\ \mathit{castStablePtrToPtr}\ \mathit{sptr}
\end{array}
$$

We can now define the *Eq* instance for *Parser*.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Eq}\ (\mathit{Parser}\ a)\ \mathbf{where}\\
\quad p \equiv q = \mathbf{case}\ p\ `\mathit{eq}`\ q\ \mathbf{of}\\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{False}\\
\qquad\qquad\qquad\qquad \mathit{Just\ Eq} \rightarrow \mathit{True}
\end{array}
$$

Left factorization

We want to be able to merge states that have common parsers in their stack.
First we modify our transition function to be able to unfold and merge states

$$
\begin{array}{l}
\mathit{transition} :: \mathit{States}\ a \to \mathit{Char} \to \mathit{States}\ a \\
\mathit{transition}\ \mathit{states}\ \mathit{char} = \\
\quad \mathbf{let}\ \mathit{unFoldNtStates} = \mathit{unFoldAndMerge}\ \mathit{states} \\
\quad \mathbf{in}\ \ \mathit{foldl}\ (\mathit{reduce}\ \mathit{char})\ [\,]\ \mathit{unFoldNtStates}
\end{array}
$$

Left factorization

$$
\begin{aligned}
&\mathit{unFoldAndMerge} :: \mathit{States}\ a \rightarrow \mathit{States}\ a\\
&\mathit{unFoldAndMerge}\ \mathit{states}\\
&\quad \mid \mathit{termination}\ \mathit{states} = \mathit{states}\\
&\quad \mid \mathit{otherwise} = \mathit{unFoldAndMerge}\ \$\ \mathit{unFoldNtAtHeads}\ \mathit{states}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{termination} :: \mathit{States}\ a \rightarrow \mathit{Bool}\\
&\mathit{termination}\ \mathit{xs} = \mathit{all}\ \mathit{isFolded}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isFolded} :: \mathit{State}\ a \rightarrow \mathit{Bool}\\
&\mathit{isFolded}\ (\mathit{State}\ r\ \mathit{Done}) && = \mathit{True}\\
&\mathit{isFolded}\ (\mathit{State}\ r\ (\mathit{Stack}\ (\mathit{Sym}\ c)\ \mathit{rest})) && = \mathit{True}\\
&\mathit{isFolded}\ \_ && = \mathit{False}
\end{aligned}
$$

Left factorization

To be able to merge we need to add new constructors to our *Pending* structure

$$Split :: Pending\ r1 \rightarrow Pending\ r2 \rightarrow Pending\ (Either\ r1\ r2)$$

and generalize our semantic function

$$\textbf{data}\ State\ a = \forall b \circ State\ (Func\ b\ a)\ (Pending\ b)$$

$$\begin{array}{l} \textbf{data}\ Func\ b\ a\ \textbf{where} \\ \quad Single :: (b \rightarrow a) \qquad\qquad\qquad\quad \rightarrow Func\ b\ a \\ \quad Two \quad :: (b \rightarrow a) \rightarrow (b \rightarrow a) \rightarrow Func\ b\ a \end{array}$$

Left factorization

To be able to merge we need to add new constructors to our *Pending* structure

$$Split :: Pending\ r1 \rightarrow Pending\ r2 \rightarrow Pending\ (Either\ r1\ r2)$$

and generalize our semantic function

$$\begin{array}{l}
\textbf{data}\ State\ a = \forall b \circ State\ (Func\ b\ a)\ (Pending\ b) \\
\textbf{data}\ Func\ b\ a\ \textbf{where} \\
\quad Single :: (b \rightarrow a) \qquad\qquad\qquad\quad \rightarrow Func\ b\ a \\
\quad Two \quad :: (b \rightarrow a) \rightarrow (b \rightarrow a) \rightarrow Func\ b\ a
\end{array}$$

Left factorization

We need to modify *unFoldNtAtHeads* to implement sharing

```
unFoldNtAtHeads :: States a → States a
unFoldNtAtHeads [] = []
```

```
unFoldNtAtHeads (s@(State func (Stack (Ret f) rest)) : states) =
   let ns = unFoldRetAtHead s
   in  unFoldNtAtHeads $ ns : states
```

```
unFoldRetAtHead :: State a → State a
unFoldRetAtHead (State (Single r) (Stack (Ret f) rest)) =
   State (Single (λrest → r (f, rest))) rest
unFoldRetAtHead (State (Two f1 f2) (Stack (Ret f) rest)) =
   State (Two (λr1 → f1 (f, r1)) (λr2 → f2 (f, r2))) rest
```

Left factorization

We need to modify *unFoldNtAtHeads* to implement sharing

$$
\begin{aligned}
&unFoldNtAtHeads :: States\ a \rightarrow States\ a \\
&unFoldNtAtHeads\ [\,] = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&unFoldNtAtHeads\ (s@(State\ func\ (Stack\ (Ret\ f)\ rest)) : states) = \\
&\quad \mathbf{let}\ ns = unFoldRetAtHead\ s \\
&\quad \mathbf{in}\ \ unFoldNtAtHeads\ \$\ ns : states
\end{aligned}
$$

$$
\begin{aligned}
&unFoldRetAtHead :: State\ a \rightarrow State\ a \\
&unFoldRetAtHead\ (State\ (Single\ r)\ (Stack\ (Ret\ f)\ rest)) = \\
&\quad State\ (Single\ (\lambda rest \rightarrow r\ (f, rest)))\ rest \\
&unFoldRetAtHead\ (State\ (Two\ f1\ f2)\ (Stack\ (Ret\ f)\ rest)) = \\
&\quad State\ (Two\ (\lambda r1 \rightarrow f1\ (f, r1))\ (\lambda r2 \rightarrow f2\ (f, r2)))\ rest
\end{aligned}
$$

Left factorization

We need to modify *unFoldNtAtHeads* to implement sharing

$$
\begin{array}{l}
\mathit{unFoldNtAtHeads} :: \mathit{States}\ a \rightarrow \mathit{States}\ a \\
\mathit{unFoldNtAtHeads}\ [\,] = [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{unFoldNtAtHeads}\ (s@(\mathit{State}\ \mathit{func}\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) : \mathit{states}) = \\
\quad \mathbf{let}\ \mathit{ns} = \mathit{unFoldRetAtHead}\ s \\
\quad \mathbf{in}\ \ \mathit{unFoldNtAtHeads}\ \$\ \mathit{ns} : \mathit{states}
\end{array}
$$

$$
\begin{array}{l}
\mathit{unFoldRetAtHead} :: \mathit{State}\ a \rightarrow \mathit{State}\ a \\
\mathit{unFoldRetAtHead}\ (\mathit{State}\ (\mathit{Single}\ r)\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) = \\
\quad \mathit{State}\ (\mathit{Single}\ (\lambda \mathit{rest} \rightarrow r\ (f, \mathit{rest})))\ \mathit{rest} \\
\mathit{unFoldRetAtHead}\ (\mathit{State}\ (\mathit{Two}\ f1\ f2)\ (\mathit{Stack}\ (\mathit{Ret}\ f)\ \mathit{rest})) = \\
\quad \mathit{State}\ (\mathit{Two}\ (\lambda r1 \rightarrow f1\ (f, r1))\ (\lambda r2 \rightarrow f2\ (f, r2)))\ \mathit{rest}
\end{array}
$$

Left factorization

We can now share equal choices

$$
\begin{array}{l}
\mathit{unFoldNtAtHeads}\ (s@(\mathit{State}\ \mathit{func}\ (\mathit{Stack}\ (\mathit{Alt}\ p\ q)\ \mathit{rest})) : \mathit{states}) = \\
\quad \textbf{let}\ \mathit{optState} = \mathit{unFoldAltAtHead}\ s \\
\quad \textbf{in}\ \ \mathit{optState} +\!\!+\ \mathit{unFoldNtAtHeads}\ \mathit{states}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{unFoldAlt} :: \mathit{Parser}\ a \rightarrow [\mathit{Parser}\ a] & \\
\mathit{unFoldAlt}\ (\mathit{Alt}\ p\ q) & = \mathit{unFoldAlt}\ p +\!\!+\ \mathit{unFoldAlt}\ q \\
\mathit{unFoldAlt}\ p & = [p]
\end{array}
$$

$$
\begin{array}{l}
\mathit{unFoldAltAtHead} :: \mathit{State}\ a \rightarrow \mathit{States}\ a \\
\mathit{unFoldAltAtHead}\ (\mathit{State}\ r\ (\mathit{Stack}\ \mathit{altParser}\ \mathit{rest})) = \\
\quad \textbf{let}\ \mathit{lstAltParsers} \qquad\ = \mathit{unFoldAlt}\ \mathit{altParser} \\
\qquad\quad \mathit{uniqueAltParsers} = \mathit{nub}\ \mathit{lstAltParsers} \\
\quad \textbf{in}\ \ \mathit{map}\ (\lambda p \rightarrow \mathit{State}\ r\ (\mathit{Stack}\ p\ \mathit{rest}))\ \mathit{uniqueAltParsers}
\end{array}
$$

Left factorization

The interesting case and real value of sharing occurs when we sequence

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (s@(\mathit{State}\ \mathit{func}\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ \mathit{rest})) : \mathit{states}) =\\
&\quad \mathbf{let}\ (\mathit{optState}, \mathit{nst}) = \mathit{unFoldSeqAtHead}\ s\ \mathit{states}\\
&\quad\quad\quad \mathit{optStates} = \mathit{unFoldNtAtHeads}\ \mathit{nst}\\
&\quad \mathbf{in}\ \ \mathit{optState} : \mathit{optStates}
\end{aligned}
$$

Left factorization

The interesting case and real value of sharing occurs when we sequence

$$
\begin{array}{l}
\mathit{unFoldSeqAtHead} :: \mathit{State}\ a \rightarrow \mathit{States}\ a \rightarrow (\mathit{State}\ a, \mathit{States}\ a) \\
\mathit{unFoldSeqAtHead}\ v@(\mathit{State}\ (\mathit{Single}\ r)\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ r1)) \\
\quad (k@(\mathit{State}\ (\mathit{Single}\ s)\ (\mathit{Stack}\ (\mathit{Seq}\ t\ u)\ r2)) : \mathit{states}) = \\
\quad \mathbf{case}\ p\ `\mathit{eq}`\ t\ \mathbf{of} \\
\qquad \mathit{Just}\ \mathit{Eq} \rightarrow \mathbf{let}\ \mathit{opts} = \mathit{State} \\
\qquad\qquad\qquad\qquad (\mathit{Two}\ (\lambda(\mathit{pr}, \mathit{Left}\ \ (\mathit{qr}, r1)) \rightarrow r\ (\mathit{pr}\ \mathit{qr}, r1)) \\
\qquad\qquad\qquad\qquad\qquad (\lambda(\mathit{pr}, \mathit{Right}\ (\mathit{qu}, r2)) \rightarrow s\ (\mathit{pr}\ \mathit{qu}, r2)) \\
\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Split}\ (\mathit{Stack}\ q\ r1)\ (\mathit{Stack}\ u\ r2))) \\
\qquad\qquad\qquad \mathbf{in}\ \ (\mathit{opts}, \mathit{states}) \\
\qquad \mathit{Nothing} \rightarrow \mathbf{let}\ (\mathit{nst}, \mathit{nsts}) = \mathit{unFoldSeqAtHead}\ v\ \mathit{states} \\
\qquad\qquad\qquad \mathbf{in}\ (\mathit{nst}, k : \mathit{nsts})
\end{array}
$$

Left factorization

Once we encounter *Split* we split the current state

$$
\begin{aligned}
&\mathit{unFoldNtAtHeads}\ (s@(\mathit{State}\ (\mathit{Two}\ \mathit{fl}\ \mathit{fr})\ (\mathit{Split}\ \mathit{pl}\ \mathit{pr})) : \mathit{states}) = \\
&\quad \textbf{let}\ \mathit{sleft}\ \ = \mathit{State}\ (\mathit{Single}\ (\lambda r1 \rightarrow \mathit{fl}\ (\mathit{Left}\ \ r1)))\ \mathit{pl} \\
&\quad \phantom{\textbf{let}}\ \mathit{sright} = \mathit{State}\ (\mathit{Single}\ (\lambda r2 \rightarrow \mathit{fr}\ (\mathit{Right}\ r2)))\ \mathit{pr} \\
&\quad \textbf{in}\ \ \mathit{sleft} : \mathit{sright} : \mathit{states}
\end{aligned}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Left factorization

$$f = (\lambda x\ z \rightarrow x : z : [\,]) <\$> Sym\ \texttt{'x'}$$
$$s = f <*> Sym\ \texttt{'y'}$$
$$<|>$$
$$f <*> Sym\ \texttt{'z'}$$

[Stack s Done]
⇝ [Stack (f < ∗ > Sym 'y') Done, Stack (f < ∗ > Sym 'z') Done]
⇝ [Stack f (Split (Stack (Sym 'y') Done) (Stack (Sym 'z') Done))]
⇝ [Split (Stack (Sym 'y') Done) (Stack (Sym 'z') Done))]
⇝ [Stack (Sym 'y') Done, Stack (Sym 'z') Done]
⇝ [Done]

Going further - LazyShare

$$
\begin{array}{l}
\mathit{unFoldSeqAtHead} :: \mathit{State}\ a \to \mathit{States}\ a \to (\mathit{State}\ a, \mathit{States}\ a) \\
\mathit{unFoldSeqAtHead}\ v@(\mathit{State}\ (\mathit{Single}\ r)\ (\mathit{Stack}\ (\mathit{Seq}\ p\ q)\ r1)) \\
\quad (k@(\mathit{State}\ (\mathit{Single}\ s)\ (\mathit{Stack}\ (\mathit{Seq}\ t\ u)\ r2)) : \mathit{states}) = \\
\quad \mathbf{case}\ p\ `\mathit{eq}`\ t\ \mathbf{of} \\
\qquad \mathit{Just}\ \mathit{Eq} \to \mathbf{let}\ \mathit{opts} = \mathit{State} \\
\qquad\qquad\qquad\qquad (\mathit{Two}\ (\lambda(\mathit{pr}, \mathit{Left}\ \ (\mathit{qr}, r1)) \to r\ (\mathit{pr}\ \mathit{qr}, r1)) \\
\qquad\qquad\qquad\qquad\qquad\quad (\lambda(\mathit{pr}, \mathit{Right}\ (\mathit{qu}, r2)) \to s\ (\mathit{pr}\ \mathit{qu}, r2)) \\
\qquad\qquad\qquad\qquad (\mathit{Stack}\ p\ (\mathit{Split}\ (\mathit{Stack}\ q\ r1)\ (\mathit{Stack}\ u\ r2))) \\
\qquad\qquad\quad \mathbf{in}\ \ (\mathit{opts}, \mathit{states}) \\
\qquad \mathit{Nothing} \to \mathbf{let}\ (\mathit{nst}, \mathit{nsts}) = \mathit{unFoldSeqAtHead}\ v\ \mathit{states} \\
\qquad\qquad\quad \mathbf{in}\ (\mathit{nst}, k : \mathit{nsts})
\end{array}
$$

Here we can also check for the equality of q and u and create a state where we represent future sharing between that state and another.

Going further - LazyShare

Extend *Pending* again

$$LazyShare :: Parser\ p \rightarrow Pending\ r \rightarrow Pending\ (Share\ p\ r)$$

We need to semantically distinguish between *Stack* and *LazyShare*

$$\begin{aligned}&\mathbf{data}\ Share\ p\ r\ \mathbf{where}\\&\quad Share :: p \rightarrow r \rightarrow Share\ p\ r\end{aligned}$$

Going further - LazyShare

$$
\begin{aligned}
&xOrz = Sat\ ((\equiv \texttt{'x'}) \vee (\equiv \texttt{'z'}))\\
&s = f <\$> xOrz <*> q <*> r\\
&\ <|>\\
&\qquad g <\$> Sym\ \texttt{'x'} <*> q <*> s
\end{aligned}
$$

Going further - LazyShare

```
  [Stack s Done]
> [Stack (f <$> xOrz <*> q <*> r) Done,
   Stack (g <$> Sym ’x’ <*> q <*> s) Done]
> [Stack (f <$> xOrz <*> q) (Stack r Done),
   Stack (g <$> Sym ’x’ <*> q) (Stack s Done)]
> [Stack (f <$> xOrz) (Sharing q (Stack r Done)),
   Stack (g  <$> Sym ’x’) (Sharing q (Stack s Done))]
> [Stack f (Stack xOrz (Sharing q (Stack r Done))),
   Stack g (Stack (Sym ’x’) (Sharing q (Stack s Done
> [Stack xOrz (Sharing q (Stack r Done)),
   Stack (Sym ’x’) (Sharing q (Stack s Done))]
> [Sharing q (Stack r Done), Sharing q (Stack s Done
> [Stack q (Split (Stack r Done) (Stack s Done))]
> [Split (Stack r Done) (Stack s Done)]
> [Stack r Done, Stack s Done]
> [Done]
```

Generalization

So far we have achieve sharing between two branches of choice, however generalizing it to N branches should be easily done as long as we can map a function in *Func* to a *Pending*.

$$Split :: Pending\ r1 \rightarrow Pending\ r2 \rightarrow Pending\ (Pair\ r1\ r2)$$

$$\begin{array}{l} \textbf{data}\ Pair\ r1\ r2\ \textbf{where} \\ \quad Pair :: r1 \rightarrow r2 \rightarrow Pair\ r1\ r2 \end{array}$$

Our new State representation could be

$$\textbf{data}\ State\ a = \forall b \circ State\ [(b \rightarrow a)]\ (Pending\ b)$$

Roadmap

Motivation

YAPL - Yet Another Parsing Library

Memoisation

Conclusion

Conclusion

Advantages

- Sharing
- Natural languages grammars
- Full parsing control (online parsing, error reporting)
- Easy to use
- *Almost* well typed

Disadvantages

- Inefficiency
- Parser equality
- Some combinators might be difficult to implement efficiently

Conclusion

Advantages

- Sharing
- Natural languages grammars
- Full parsing control (online parsing, error reporting)
- Easy to use
- *Almost* well typed

Disadvantages

- Inefficiency
- Parser equality
- Some combinators might be difficult to implement efficiently

Future work

- Generalization for N branches
- Results
- Try other approaches to parser equality
- Left recursive grammars
- Monadic parsers