Universiteit Utrecht

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

Lectures C: Parser Combinators - Introduction

Doaitse Swierstra

Utrecht University

August 29, 2011

Elementary Parser Combinators

What are parser combinators

Elementary Combinators

The "List of Successes" Implementation

Developing an Embdded Domain Specific Language

Monadic Parsers

Problems

- Problems with Erroneous Input
- Problems with Left-Recursion
- Problems with Space Consumption

1. Elementary Parser Combinators

1.1 What are parser combinators

What are parser combinators

§1.1

- a collection basic parsing functions that recognise a piece of input
- a collection of combinators that build new parsers out of existing ones

What are parser combinators

§1.1

- a collection basic parsing functions that recognise a piece of input
- a collection of combinators that build new parsers out of existing ones

For the time being parsers are just Haskell functions, but we will extend on that later.

1.2 Elementary Combinators

Elementary Parsers

§1.2

All libraries are based on at least the following four basic combinators:

Elementary Parsers

All libraries are based on at least the following four basic combinators:

	Haskell
alternative	p <|> q
composition	p <*> q
terminals	*pSym* `'s'`
empty string	*pSucceed*

Building Parsers out of Parsers

§1.2

Other operators:

Building Parsers out of Parsers

Other operators:

		Haskell
repetition	$a*$	*pList a*
option	$a?$	‘*opt*‘ *a*
grouping	$(\ldots)$	$(\ldots)$

The Type of a Parser

A parser takes a string and returns result of type a

$$String \rightarrow a$$

The Type of a Parser

§1.2

A parser may be ambiguous, and there may be many possible ways of recognizing a value of type a:

> $String \rightarrow [a]$

The Type of a Parser

A parser may not consume the whole input, and now also returns the unused part of the input:

$$\mathit{String} \rightarrow [(a, \mathit{String})]$$

The Type of a Parser

§1.2

Why should we only accept character strings? Everything goes, as long as we have equality defined for the tokens:

$$Eq\ s \Rightarrow [s] \rightarrow [(a, [s])]$$

The Type of a Parser

Why should we only accept character strings? Everything goes, as long as we have equality defined for the tokens:

$$Eq\ s \Rightarrow [s] \rightarrow [(a, [s])]$$

Which we give a name:

$$\textbf{type}\ Parser\ s\ a = Eq\ s \Rightarrow [s] \rightarrow [(a, [s])]$$

Types of the Elementary Combinators

Types

$$<|> \quad :: Parser\ s\ a \quad \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a$$

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

$$
\begin{array}{lll}
<|> & :: Parser\ s\ a & \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\
<*> & :: Parser\ s\ a & \rightarrow Parser\ s\ b \rightarrow Parser\ s\ (a, b)
\end{array}
$$

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

$$
\begin{aligned}
<|> &:: Parser\ s\ a \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\
<*> &:: Parser\ s\ (b \rightarrow a) \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a
\end{aligned}
$$

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

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

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

$$
\begin{array}{lll}
<|> & :: Parser\ s\ a & \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\
<*> & :: Parser\ s\ (b \rightarrow a) & \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a \\
pSym & :: s & \rightarrow Parser\ s\ s \\
pSucceed & :: a & \rightarrow Parser\ s\ a
\end{array}
$$

Try to remember these types. Knowing the types is half the work when programming in Haskell.

Types of the Elementary Combinators

Types

$$
\begin{array}{lll}
<|> & :: Parser\ s\ a & \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\
<*> & :: Parser\ s\ (b \rightarrow a) & \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a \\
pSym & :: s & \rightarrow Parser\ s\ s \\
pSucceed & :: a & \rightarrow Parser\ s\ a \\
pFail & :: & Parser\ s\ a
\end{array}
$$

Try to remember these types. Knowing the types is half the work when programming in Haskell.

1.3 The "List of Successes" Implementation

The "List of Successes" Implementation

§1.3

$$pFail \qquad inp = [\,]$$

The "List of Successes" Implementation

$$
\begin{aligned}
\mathit{pFail} \quad & \mathit{inp} = [\,] \\
\mathit{pSucceed}\ v\ \mathit{inp} & = [\,(v, \mathit{inp})\,]
\end{aligned}
$$

The "List of Successes" Implementation

§1.3

$$
\begin{aligned}
&\mathit{pFail} && \mathit{inp} = [\,] \\
&\mathit{pSucceed}\ v && \mathit{inp} = [\,(v, \mathit{inp})\,] \\
&(p <|> q) && \mathit{inp} = p\ \mathit{inp} +\!\!+\ q\ \mathit{inp}
\end{aligned}
$$

The "List of Successes" Implementation

$$
\begin{array}{lll}
\mathit{pFail} & \mathit{inp} & = [\,] \\
\mathit{pSucceed}\ v & \mathit{inp} & = [(v, \mathit{inp})] \\
(p <|> q) & \mathit{inp} & = p\ \mathit{inp} +\!\!+ q\ \mathit{inp} \\
\mathit{pSym}\ a & (s : ss) \mid a \equiv s & = [(s, ss)] \\
\mathit{pSym}\ a & \_ & = [\,]
\end{array}
$$

The "List of Successes" Implementation

§1.3

$$
\begin{array}{lll}
\mathit{pFail} & \mathit{inp} & = [\,] \\
\mathit{pSucceed}\ v & \mathit{inp} & = [(v, \mathit{inp})] \\
(p <|> q) & \mathit{inp} & = p\ \mathit{inp} +\!\!+\ q\ \mathit{inp} \\
\mathit{pSym}\ a & (s : ss) \mid a \equiv s & = [(s, ss)] \\
\mathit{pSym}\ a & \_ & = [\,] \\
(p <\!*\!> q) & \mathit{inp} & = [(a2b\ a, rr) \mid (a2b, r) \leftarrow p\ \mathit{inp} \\
 & & \quad\quad\quad\quad\quad\quad\quad\quad , (a, \quad rr) \leftarrow q\ r \\
 & & \quad\quad\quad\quad\quad\quad\quad\quad]
\end{array}
$$

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

Computing a Result

§1.3

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

$$pSym\ \texttt{'B'}$$

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

$$pSym\ \texttt{'B'}$$

Preceded by the recognition of a character `'A'`

$$pSym\ \texttt{'A'} \qquad pSym\ \texttt{'B'}$$

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

$$pSym\ \texttt{'B'}$$

Preceded by the recognition of a character `'A'`

$$pSym\ \texttt{'A'} \qquad pSym\ \texttt{'B'}$$

We now insert a dummy parser that returns the function $(,)$:

$$pSucceed\ (,) \qquad pSym\ \texttt{'A'} \qquad pSym\ \texttt{'B'}$$

Computing a Result

The question which arises now is how do we get something useful out of such parsers?

We recognize a character `'B'`:

$$pSym\ \texttt{'B'}$$

Preceded by the recognition of a character `'A'`

$$pSym\ \texttt{'A'} \qquad pSym\ \texttt{'B'}$$

We now insert a dummy parser that returns the function $(,)$:

$$pSucceed\ (,) \qquad pSym\ \texttt{'A'} \qquad pSym\ \texttt{'B'}$$

Combine the result using sequential composition of parsers:

$$pAB = \quad pSucceed\ (,) \mathbin{<\!*\!>} pSym\ \texttt{'A'} \mathbin{<\!*\!>} pSym\ \texttt{'B'}$$

Capturing the essence of $\langle * \rangle$

Suppose we want to deal with possibly failing notations and stay as closely as possible to the original notation; how to we deal with functions applications like $e_1\ e_2$.

- both the function part e_1 and the argument part e_2 can fail to compute something
- we model this with a *Maybe*
- so we want to "apply" a *Maybe* $(b \to a)$ to a *Maybe b*, and produce a *Maybe a*

```
func `applyTo` arg = case func of
                       Just b2a → case arg of
                                    Just b    → Just (b2a b)
                                    Nothing   → Nothing
                       Nothing → Nothing
```

Capturing the essence of $<*>$ (Cont)

We capture this pattern as follows:

```
class Applicative p where
  (<*>)   :: p (b→a)→p b→p a
  pure    :: a              →p a
  (<$>)   :: (b→a)   →p b→p a
  f <$> p = pure f <*> p
  ...
instance Applicative Maybe where
  Just f <*> Just v = Just (f v)
  _      <*> _      = Nothing
```

Capturing the essence of $\langle * \rangle$ (Cont)

If we now write:

$$f \langle * \rangle a_1 \langle * \rangle a_2 \langle * \rangle a_3$$

we have "overloaded" the original implicit function applications in $f\ a_1\ a_2\ a_3$.

Capturing the essence of $\langle * \rangle$ (Cont)

If we now write:

$$f \langle * \rangle a_1 \langle * \rangle a_2 \langle * \rangle a_3$$

we have "overloaded" the original implicit function applications in $f\ a_1\ a_2\ a_3$. **Conclusion:**
Instead applying a value of type $b \rightarrow a$ to a value of type b to result in a value of type a the operator $\langle * \rangle$ applies a p-value labelled with type $b \rightarrow a$ to a p-value labelled with type b to result in a p-value labelled with type a.

All *Monad*s are *Applicative*

Every type constructor which is in the class *Monad* can be trivially made an instance of the class *Applicative*:

```
instance Monad m ⇒ Applicative m where
  f <*> v = do ff ← f
               vv ← v    -- no reference to fv in rhs
               return (ff vv)
  pure    = return
```

Advice

The essential difference is that when using the class *Applicative* we abstain from the possibility to refer to the f-value in the second binding of the **do**-construct.

Advice

The essential difference is that when using the class *Applicative* we abstain from the possibility to refer to the f-value in the second binding of the **do**-construct.
*Applicative*is to be preferred over *Monad*, since it allows optimisations; the second part is independent of the first part and can thus be evaluated "more statically", or even analysed independent of the run of the program!

Alternative

The companion class for *Applicative* is *Alternative*:

```
class Alternative m where
  (<|>) :: m a→m a→m a
  empty ::          m a
instance Alternative Maybe where
  Just l <|> _ = Just l
  _      <|> r = r
  empty        = Nothing
```

Alternative

The companion class for *Applicative* is *Alternative*:

```
class Alternative m where
  (<|>) :: m a→m a→m a
  empty ::                m a
instance Alternative Maybe where
  Just l <|> _ = Just l
  _      <|> r = r
  empty        = Nothing
```

Attention: For the **instance** *Alternative* (*Parser s*) the value *empty* is not the parser which recognises the empty string, but the parser that always fails!

1.4 Developing an Embedded Domain Specific Language

Useful functions I

Because the pattern:

$$pSucceed\ f \mathbin{<\!*\!>} p$$

occurs so often

Useful functions I

Because the pattern:

$$pSucceed\ f\ <*>\ p$$

occurs so often we define

$<\$>$

$$f\ <\$>\ p = pSucceed\ f\ <*>\ p$$

Useful functions I

Because the pattern:

$$pSucceed\ f\ <*>\ p$$

occurs so often we define

$<\$>$

$$f\ <\$>\ p = pSucceed\ f\ <*>\ p$$

so we can write the previous function as:

$$pAB = (,)\ <\$>\ pSym\ \texttt{'A'}\ <*>\ pSym\ \texttt{'B'}$$

Useful functions II

Often we are not interested in parts of what we have recognized:

$$
\begin{aligned}
&\mathit{semIfStat}\ \mathit{cond}\ \mathit{ifpart}\ \mathit{thenpart} = \ldots \\
&\mathit{pIfStat} = (\lambda\_\ c\ \_\ t\ \_\ e\ \_ \quad \rightarrow \mathit{semIfStat}\ c\ t\ e) \\
&\qquad\qquad <\$> \mathit{pIfToken} \quad <*> \mathit{pExpr} \\
&\qquad\qquad <*> \mathit{pThenToken} <*> \mathit{pExpr} \\
&\qquad\qquad <*> \mathit{pElseToken} <*> \mathit{pExpr} \\
&\qquad\qquad <*> \mathit{pFiToken}
\end{aligned}
$$

Useful functions II

Often we are not interested in parts of what we have recognized:

$$
\begin{array}{l}
\mathit{semIfStat}\ \mathit{cond}\ \mathit{ifpart}\ \mathit{thenpart} = \ldots \\
\mathit{pIfStat} = (\lambda\_\ c\ \_\ t\ \_\ e\ \_ \quad \rightarrow \mathit{semIfStat}\ c\ t\ e) \\
\qquad\qquad <\$> \mathit{pIfToken} \quad <*> \mathit{pExpr} \\
\qquad\qquad <*> \mathit{pThenToken} <*> \mathit{pExpr} \\
\qquad\qquad <*> \mathit{pElseToken} <*> \mathit{pExpr} \\
\qquad\qquad <*> \mathit{pFiToken}
\end{array}
$$

We define

$$
\begin{array}{l}
p <*\ \ q = (\lambda x\ \_ \rightarrow x) <\$> p <*> q \\
p\ \ *> q = (\lambda\_\ y \rightarrow y) <\$> p <*> q \\
f <\$\ \ q = \mathit{pSucceed}\ f <*\ \ q
\end{array}
$$

Useful functions II

We define

$$
\begin{aligned}
p \mathbin{<\!*} q &= (\lambda x \_ \to x) \mathbin{<\$>} p \mathbin{<\!*\!>} q \\
p \mathbin{*\!>} q &= (\lambda \_\, y \to y) \mathbin{<\$>} p \mathbin{<\!*\!>} q \\
f \mathbin{<\$} q &= \mathit{pSucceed}\, f \mathbin{<\!*} q
\end{aligned}
$$

So we can now write:

$$
\begin{aligned}
\mathit{pIfStat} = \mathit{semIfStat} \mathbin{<\$} \; & \mathit{pIfToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pThenToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pElseToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pFiToken}
\end{aligned}
$$

Useful functions II

We define

$$
\begin{aligned}
p \mathbin{<\!*} q &= (\lambda x \_ \to x) \mathbin{<\$>} p \mathbin{<\!*\!>} q \\
p \mathbin{*\!>} q &= (\lambda \_\, y \to y) \mathbin{<\$>} p \mathbin{<\!*\!>} q \\
f \mathbin{<\$} q &= \mathit{pSucceed}\, f \mathbin{<\!*} q
\end{aligned}
$$

So we can now write:

$$
\begin{aligned}
\mathit{pIfStat} = \mathit{semIfStat} \mathbin{<\$} \; & \mathit{pIfToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pThenToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pElseToken} \mathbin{<\!*\!>} \mathit{pExpr} \\
\mathbin{<\!*} \; & \mathit{pFiToken}
\end{aligned}
$$

Functions like *semIfStat* are generated by the *uuagc* compiler.

EBNF extensions


```
infixl 2 opt
opt          :: Parser s a→a→Parser s a
p `opt` v = p <|> pSucceed v
```

In the library we have special greedy versions which choose the longest alternative.

EBNF extensions


```
infixl 2 opt
opt          :: Parser s a→a→Parser s a
p `opt` v = p <|> pSucceed v
```

```
pList    :: Parser s a→Parser s [a]
pList p = (:) <$> p <*> pList p `opt` []
```

In the library we have special greedy versions which choose the longest alternative.

Exercise

Write a function that recognises a sequence of balanced parentheses, (i.e. $(), (()), (()\ ())\ (), \ldots$, and computes the maximal nesting depth (here $1, 2, 2, \ldots$. The grammar describing this language is:

$$S \rightarrow (S)\ S \mid \cdot$$

$$pP = (max \cdot (+1)) <\$\ \ pSym\ \text{'('}\ <*>\ p <*\ \ pSym\ \text{')'}\ <*>\ p$$

$$\text{`}opt\text{`}$$

$$0$$

Chaining

$$
\begin{aligned}
&\mathit{pChainr} :: \mathit{Parser}\ s\ (c \to c \to c) \to \mathit{Parser}\ s\ c \to \mathit{Parser}\ s\ c \\
&\mathit{pChainr}\ \mathit{sep}\ p = p \\
&\qquad\qquad <|> (\lambda \mathit{pres}\ \mathit{op}\ \mathit{chres} \to \mathit{pres}\ \text{`}\mathit{op}\text{`}\ \mathit{chres}) \\
&\qquad\qquad\qquad <\$> p <*> \mathit{sep} <*> \mathit{pChainr}\ \mathit{sep}\ p
\end{aligned}
$$

Chaining


```
pChainr :: Parser s (c→c→c)→Parser s c→Parser s c
pChainr sep p = p
                <|> (λpres op chres→pres `op` chres)
                    <$> p <*> sep <*> pChainr sep p
```

```
pChainl :: Parser s (c→c→c)→Parser s c→Parser s c
pChainl op x = (f <$> x <*> pList (flip <$> op <*> x))
               where
               f x [] = x
               f x (func : rest) = f (func x) rest
```

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same element:

$$
\begin{aligned}
p = \quad & f \mathbin{<\$>} q \mathbin{<*>} r1 \\
\mathbin{<|>}\ & g \mathbin{<\$>} q \mathbin{<*>} r2
\end{aligned}
$$

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same element:

$$
\begin{aligned}
p = \quad & f \mathbin{<\$>} q \mathbin{<*>} r1 \\
\mathbin{<|>}\ & g \mathbin{<\$>} q \mathbin{<*>} r2
\end{aligned}
$$

So we define:

$$
\begin{aligned}
& p \mathbin{<**>} q :: \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ a \\
& p \mathbin{<**>} q = (\lambda pv\ qv \to qv\ pv) \mathbin{<\$>} p \mathbin{<*>} q
\end{aligned}
$$

Left Factorisation

It is not a good idea to have parsers that have alternatives starting with the same element:

$$
\begin{aligned}
p = \quad & f \mathbin{<\$>} q \mathbin{<*>} r1 \\
\mathbin{<|>}\ & g \mathbin{<\$>} q \mathbin{<*>} r2
\end{aligned}
$$

So we define:

$$
\begin{aligned}
& p \mathbin{<**>} q :: \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ a \\
& p \mathbin{<**>} q = (\lambda pv\ qv \to qv\ pv) \mathbin{<\$>} p \mathbin{<*>} q
\end{aligned}
$$

So we can replace the above code by:

$$
\begin{aligned}
& p = q \mathbin{<**>} (\mathit{flip}\ f \mathbin{<\$>} r1 \mathbin{<|>} \mathit{flip}\ g \mathbin{<\$>} r2) \\
& \mathit{flip}\ f\ x\ y = f\ y\ x
\end{aligned}
$$

More Useful Combinators

$<??>$

$$p <??> q :: Parser\ s\ a \rightarrow Parser\ s\ (a \rightarrow a) \rightarrow Parser\ s\ a$$
$$p <??> q = p <**> (q\ `opt`\ id)$$

More Useful Combinators

$<??>$

$$p <??> q :: Parser\ s\ a \to Parser\ s\ (a \to a) \to Parser\ s\ a$$
$$p <??> q = p <**> (q\ `opt`\ id)$$

$pParens$

$$pParens :: s \to s \to Parser\ s\ a \to Parser\ s\ a$$
$$pParens\ l\ r = pSym\ l\ *> x <*\ pSym\ r$$

Left Factorisation II

We want to recognise expressions with as result a value of the type:

$$
\begin{array}{lllll}
\textbf{data}\ \mathit{Expr} = & \mathit{Lambda} & \mathit{Id} & \mathit{Expr} \\
\quad\quad\quad\quad\ \mid & \mathit{App} & \mathit{Expr} & \mathit{Expr} \\
\quad\quad\quad\quad\ \mid & \mathit{TypedExpr} & \mathit{TypeDescr} & \mathit{Expr}
\end{array}
$$

Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda     Id          Expr
          | App        Expr        Expr
          | TypedExpr  TypeDescr   Expr
```

```
pFactor = Lambda <$  pSym '\\' <*> pIdent <*> pExpr
          <|>
          pParens '(' ')' pExpr
```

Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda     Id          Expr
          | App        Expr        Expr
          | TypedExpr  TypeDescr   Expr
```

```
pFactor = Lambda <$  pSym '\\' <*> pIdent <*> pExpr
          <|>
          pParens '(' ')' pExpr
```

```
pExpr = pChainl (pSucceed App) pFactor
        <??>    (   TypedExpr
                <$  pTok "::"
                <*> pTypeDescr)
```

Chain Parsers

§1.4

A nicer definition for *pChainr* is:

$$pChainr\ sep\ p = p <??> (flip <\$> sep <*> pChainr\ sep\ p)$$

1.5 Monadic Parsers

The Chomsky Hierarchy

§1.5

The Chomsky hierarchy:

- Regular
- Context-free
- Context-sensitive
- Recursively enumerable

It is well known that context free grammars have limited expressibility.

Recognising $\{a^n b^n c^n\}$

$$
\begin{aligned}
&\mathit{times} && :: \mathit{Int} \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ [a] \\
&0\ \text{`}\mathit{times}\text{`}\ p && = \mathit{pSucceed}\ [\,] \\
&n\ \text{`}\mathit{times}\text{`}\ p && = (:)\ <\$>\ p\ <*>\ (n-1)\ \text{`}\mathit{times}\text{`}\ p
\end{aligned}
$$

Recognising $\{a^n b^n c^n\}$

$$
\begin{aligned}
&\mathit{times} && :: \mathit{Int} \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ [a] \\
&0\ \text{`}\mathit{times}\text{`}\ p && = \mathit{pSucceed}\ [\,] \\
&n\ \text{`}\mathit{times}\text{`}\ p && = (:) <\$> p <*> (n-1)\ \text{`}\mathit{times}\text{`}\ p \\
&\mathit{abc}\ n && = n <\$\ \ (n\ \text{`}\mathit{times}\text{`}\ a) \\
& && \quad\ <*\ \ (n\ \text{`}\mathit{times}\text{`}\ b) \\
& && \quad\ <*\ \ (n\ \text{`}\mathit{times}\text{`}\ c)
\end{aligned}
$$

Recognising $\{a^n b^n c^n\}$

$$
\begin{array}{lll}
times & :: & Int \to Parser\ s\ a \to Parser\ s\ [a] \\
0\ \text{`}times\text{`}\ p & = & pSucceed\ [\,] \\
n\ \text{`}times\text{`}\ p & = & (:)\ <\$>\ p\ <*>\ (n-1)\ \text{`}times\text{`}\ p \\
abc\ n & = & n\ <\$\ \ (n\ \text{`}times\text{`}\ a) \\
 & & \quad <*\ \ (n\ \text{`}times\text{`}\ b) \\
 & & \quad <*\ \ (n\ \text{`}times\text{`}\ c) \\
ABC & = & foldr\ (<|>)\ pFail\ [abc\ n \mid n \leftarrow 0\,..]
\end{array}
$$

Recognising $\{a^n b^n c^n\}$

$$
\begin{array}{lll}
times & :: Int \to Parser\ s\ a \to Parser\ s\ [a] \\
0\ \text{`}times\text{`}\ p & = pSucceed\ [\,] \\
n\ \text{`}times\text{`}\ p & = (:) <\$> p <*> (n-1)\ \text{`}times\text{`}\ p \\
abc\ n & = n <\$\ \ (n\ \text{`}times\text{`}\ a) \\
 & \quad\ \ <*\ \ (n\ \text{`}times\text{`}\ b) \\
 & \quad\ \ <*\ \ (n\ \text{`}times\text{`}\ c) \\
ABC & = foldr\ (<|>)\ pFail\ [abc\ n \mid n \leftarrow 0\,..]
\end{array}
$$

We admit that this is not very efficient, but left factorisation is not so easy since the corresponding context free grammar is infinite.

Recognising $\{a^n b^n c^n\}$: The Monadic Approach §1.5

Wouldn't it be nice if we could start by just recognizing a sequence of a's, and then use the result to enforce the right number of b's and c's?

Recognising $\{a^n b^n c^n\}$: The Monadic Approach §1.5

Wouldn't it be nice if we could start by just recognizing a sequence of a's, and then use the result to enforce the right number of b's and c's?

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ (\mathit{Parser}\ s)\ \textbf{where} \\
\quad p\ (\gg\!=)\ q = \lambda inp \to [\,(qv, rr) \mid (pv, r) \leftarrow p\ \ inp \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , (qv, rr) \leftarrow q\ pv\ r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \\
\quad return\ v\ \ = \lambda inp \to [\,(v, inp)\,]
\end{array}
$$

Recognising $\{a^n b^n c^n\}$: The Monadic Approach §1.5

Wouldn't it be nice if we could start by just recognizing a sequence of a's, and then use the result to enforce the right number of b's and c's?

```
instance Monad (Parser s) where
  p (>>=) q = λinp → [(qv, rr) | (pv, r) ← p     inp
                               , (qv, rr) ← q pv r
                               ]
  return v  = λinp → [(v, inp)]
as          :: Parser Char Int
as          = length <$ pList (pSym 'a')
bc n        = n <$ (n `times` b) <* (n `times` c)
```

Recognising $\{a^n b^n c^n\}$: The Monadic Approach §1.5

Wouldn't it be nice if we could start by just recognizing a sequence of a's, and then use the result to enforce the right number of b's and c's?

```
instance Monad (Parser s) where
  p (>>=) q = λinp → [(qv, rr) | (pv, r) ← p     inp
                               , (qv, rr) ← q pv r
                               ]
  return v  = λinp → [(v, inp)]
as          :: Parser Char Int
as          = length <$ pList (pSym 'a')
bc n        = n <$ (n `times` b) <* (n `times` c)
ABC         = do n ← as
                 bc n
```

Intersection of Languages

§1.5

Can we achieve the same effect by using :
$\{a^n b^n c^n\} = \{a^i b^j c^j\} \cap \{a^j b^j c^i\}$?

Intersection of Languages

Can we achieve the same effect by using :
$\{a^n b^n c^n\} = \{a^i b^j c^j\} \cap \{a^j b^j c^i\}$?
We redefine the type *Parser* to keep track of the length of the recognized part:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Parser}\ s\ a & = & [s] \rightarrow [(a, [s], \mathit{Int})] \\
(p \mathbin{<\!*\!>} q)\ \mathit{inp} & = & [\,(pv\ qv, rr, pc + qc) \mid (pv, r,\ pc) \leftarrow p\ \mathit{inp} \\
 & & \qquad\qquad\qquad\qquad\quad ,\ (qv, rr, qc) \leftarrow q\ r \\
 & & \,] \\
\ldots & &
\end{array}
$$

Continued ..

$$
\begin{array}{l}
(<\&\&>) :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ (a,b) \\
(p <\&\&> q)\ inp = [((pv,qv),r,pc) \mid (pv,r,pc) \leftarrow p\ inp \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad , (qv,_,qc) \leftarrow q\ inp \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad , pc \equiv qc \\
\qquad\qquad\quad]
\end{array}
$$

$$
\begin{array}{l}
jj\ x\ y = \mathbf{let}\ xy = (2+) <\$\ pSym\ x <*> xy <*\ pSym\ y\ \text{`}opt\text{`}\ 0 \\
i\ x \quad = (1+) \quad <\$\ pSym\ x <*> i\ x\ \text{`}opt\text{`}\ 0 \\
ABC = \qquad (+) <\$\ i\ \texttt{'a'} \quad <*> jj\ \texttt{'b'}\ \texttt{'c'} \\
\qquad <\&\&> (+) <\$> jj\ \texttt{'a'}\ \texttt{'b'} <*> i \quad \texttt{'c'}
\end{array}
$$

1.6 Problems

Problems with Erroneous Input

§1.6

- 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.

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.

These problem have been cured in both Parsec and the UUParsing-library. The latter does this:

- without much overhead
- without need for help from the programmer
- without stopping, so many errors can be found in a single run

Left Recursions

As with any top-down parsing method, having left-recursive parsers is no good

- You get non-terminating parsers
- You get no error messages

Left Recursions

As with any top-down parsing method, having left-recursive parsers is no good

- You get non-terminating parsers
- You get no error messages

This problem can be partially been cured as we will see later.

Problems with Space Consumption

Our naive "List of successes" implementation has further drawbacks:

- The complete input has to be parsed before any result 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 in the uu-parsinglib package.