Universiteit Utrecht

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

Lectures C: Parser Combinators - Introduction

Doaitse Swierstra

Utrecht University

August 23, 2010

Domain Specific Languages

A DSL is a programming language tailored for a particular application domain, which captures precisely the semantics of the application domain:

Domain Specific Languages

A DSL is a programming language tailored for a particular application domain, which captures precisely the semantics of the application domain:

- Lex / Yacc (lexing and parsing)
- LaTeX(for document mark-up)
- Tcl/Tk (GUI scripting)
- MatLab (numerical computations)

Domain Specific Languages

Advantages of using a DSL are that programs:

- are easier to understand
- are easier to write
- are easier to maintain
- can be written by non-programmers

Domain Specific Languages

Advantages of using a DSL are that programs:

- are easier to understand
- are easier to write
- are easier to maintain
- can be written by non-programmers

Disadvantages:

- High start-up cost
  - design and implementation of a new language is hard
- Lack of "general purpose" features (e.g. abstraction)
- Little tool support

Domain Specific Embedded Languages

Embed a DSL as a library in a general purpose host language:

- inherit general purpose features from host language
  - abstraction mechanism
  - type system
- inherit compilers and tools
- good integration with host language
- many DSL's can easily be used together!

Domain Specific Embedded Languages

There are however also disadvantages:

- Constrained by host language
  - Syntax
  - Type system
- Implementation shines through

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- Infix syntax allows you to make programs look nice

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- Infix syntax allows you to make programs look nice
- List and monad comprehensions

DSEL's in Haskell

Haskell is a very suitable host for DSEL's:

- Polymorphism
- Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- Infix syntax allows you to make programs look nice
- List and monad comprehensions
- Type classes for passing extra arguments in an implicit way

Examples

- Parser combinators
- Pretty printing libraries
- HaskellDB for implicitly generating SQL
- QuickCheck
- GUI libraries
- WASH/CGI – for describing HTML pages
- Haskore – for describing music
- Agent based systems
- Financial combinators – for describing financial products

General Purpose Programming Language

Old idea:

A general purpose programming language should make it easy to write any kind of program.

General Purpose Programming Language

Old idea:

A general purpose programming language should make it easy to write any kind of program.

New idea:

A general purpose programming language should make it possible to write very powerful libraries; even if this is a substantial effort.

A short overview

We will:

- show an example of a combinator language

A short overview

We will:

- show an example of a combinator language
- show how it can be extended

A short overview

We will:

- show an example of a combinator language
- show how it can be extended
- show how we can produce smarter implementations

A short overview

We will:

- show an example of a combinator language
- show how it can be extended
- show how we can produce smarter implementations
- show how we can emded "compilers". i.e. how we can analyse and transform the EDSL

What are parser combinators

- 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

- 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.

Elementary Parsers

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

Elementary Parsers

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

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

Building Parsers out of Parsers

Other operators:

Building Parsers out of Parsers

Other operators:

		Haskell
repetition	$a*$	$pMany\ a$
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

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

$$\mathit{String} \to [a]$$

The Type of a Parser

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

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

The Type of a Parser

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

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

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:

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

Types of the Elementary Combinators

Types

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

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

Types of the Elementary Combinators

Types

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

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

Types of the Elementary Combinators

Types

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

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

Types of the Elementary Combinators

Types

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

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

Types of the Elementary Combinators

Types

$$
\begin{aligned}
<|> &:: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
<*> &:: \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a \\
\mathit{pSym} &:: s \to \mathit{Parser}\ s\ s \\
\mathit{pure} &:: a \to \mathit{Parser}\ s\ a
\end{aligned}
$$

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

Types of the Elementary Combinators

Types

$$
\begin{aligned}
<|> &:: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
<*> &:: \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a \\
\mathit{pSym} &:: s \to \mathit{Parser}\ s\ s \\
\mathit{pure} &:: a \to \mathit{Parser}\ s\ a \\
\mathit{empty} &:: \mathit{Parser}\ s\ a
\end{aligned}
$$

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

The Class *Applicative*

```
class Functor f ⇒ Applicative f where
  pure  :: a                    → f a
  (<*>) :: f (a → b) → f a → f b
  ( *>) :: f a       → f b → f b
  (<* ) :: f a       → f b → f a
class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
  some  :: f a       → f [a]
  many  :: f a       → f [a]
```

Besides the monadic style of programming more and more libraries are based on the these classes.

Maybe is an instance of these

```
instance Applicative Maybe where
  pure      = return
  f <*> a   = do f' ← f
                 a' ← a
                 return (f' a')
instance Alternative Maybe where
  empty           = Nothing
  Nothing <|> p   = p
  Just x  <|> _   = Just x
```

For more instances see the Haddock dicumentation of these classes in *Control.Applicative*

The "List of Successes" Implementation

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

The "List of Successes" Implementation

$$
\begin{array}{ll}
\mathit{empty} & \mathit{inp} = [\,] \\
\mathit{pure}\ v & \mathit{inp} = [\,(v, \mathit{inp})\,]
\end{array}
$$

The "List of Successes" Implementation

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

The "List of Successes" Implementation

$$
\begin{array}{lll}
\mathit{empty} & \mathit{inp} & = [\,] \\
\mathit{pure}\ v & \mathit{inp} & = [\,(v, \mathit{inp})\,] \\
(p <|> q) & \mathit{inp} & = p\ \mathit{inp} +\!\!+ q\ \mathit{inp} \\
\mathit{pSym}\ a & \mathit{inp} & = \textbf{case}\ \mathit{inp}\ \textbf{of} \\
 & & \quad (s : ss) \rightarrow \textbf{if}\ \ a \equiv s \\
 & & \qquad\qquad\quad \textbf{then}\ [\,(s, ss)\,] \\
 & & \qquad\qquad\quad \textbf{else}\ [\,] \\
 & & \quad [\,] \rightarrow [\,]
\end{array}
$$

The "List of Successes" Implementation

$$
\begin{array}{lll}
\mathit{empty} & \mathit{inp} = & [\,] \\
\mathit{pure}\ v & \mathit{inp} = & [\,(v, \mathit{inp})\,] \\
(p <|> q) & \mathit{inp} = & p\ \mathit{inp} +\!\!+\ q\ \mathit{inp} \\
\mathit{pSym}\ a & \mathit{inp} = & \textbf{case}\ \mathit{inp}\ \textbf{of} \\
 & & \quad (s : ss) \rightarrow \textbf{if}\ \ a \equiv s \\
 & & \qquad\qquad\quad \textbf{then}\ [\,(s, ss)\,] \\
 & & \qquad\qquad\quad \textbf{else}\ [\,] \\
 & & \quad [\,] \rightarrow [\,] \\
(p <\!*\!> q) & \mathit{inp} = & [\ (a2b\ a,\ rr) \\
 & & \ \mid (a2b\ \ ,\ r) \leftarrow p\ \mathit{inp} \\
 & & ,\ (a\ \ ,\ rr) \leftarrow q\ r \\
 & &]
\end{array}
$$

Computing a Result

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

Computing a Result

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

We recognize a character `'B'`:

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

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'`

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

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'`

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

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

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

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'`

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

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

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

Combine the result using sequential composition of parsers:

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

Useful functions I

Because the pattern:

$$pure\ f\ <\!*\!>\ p$$

occurs so often

Useful functions I

Because the pattern:

$$pure\ f\ <\!*\!>\ p$$

occurs so often we define

<$>

$$f\ <\$>\ p = pure\ f\ <\!*\!>\ p$$

Useful functions I

Because the pattern:

$$pure\ f\ <\!*\!>\ p$$

occurs so often we define

<$>

$$f\ <\$>\ p = pure\ f\ <\!*\!>\ p$$

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:

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

Useful functions II

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

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

We define

$$
\begin{aligned}
p <*\ \ q &= (\lambda x\ \_ \rightarrow x) <\$> p <*> q \\
p\ \ *> q &= (\lambda\_\ y \rightarrow y) <\$> p <*> q \\
f <\$\ \ q &= \mathit{pure}\ f <*\ \ q
\end{aligned}
$$

Useful functions II

We define

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

So we can now write:

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

Useful functions II

We define

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

So we can now write:

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

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 <|> pure 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 <|> pure 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 $1, 2, 2, \ldots$.

Exercise

Write a function that recognises a sequence of balanced parentheses, (i.e. $(), (()), (()\ ())\ (), \ldots$, and computes the maximal nesting depth (here $1, 2, 2, \ldots$.

$$
\begin{aligned}
pP &= (max \cdot (+1)) <\$> pPack\ pP <*> pP \\
&\quad \text{`}opt\text{`}\ 0 \\
pPack\ p &= pSym\ \text{'('} \ *> p <* \ pSym\ \text{')'}
\end{aligned}
$$

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

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

$$
\begin{aligned}
&pChainl :: Parser\ s\ (c \to c \to c) \to Parser\ s\ c \to Parser\ s\ c \\
&pChainl\ op\ x = (f <\$> x <*> pList\ (flip <\$> op <*> x)) \\
&\qquad \textbf{where} \\
&\qquad f\ x\ [\,] = x \\
&\qquad f\ x\ (func : rest) = f\ (func\ x)\ rest
\end{aligned}
$$

Left Factorisation

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

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

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

$<??>$

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

More Useful Combinators

$<??>$

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

$pParens$

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

Left Factorisation II

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

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

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 (pure App) pFactor
        <??> (     TypedExpr
                <$  pTok "::"
                <*> pTypeDescr
             )
```

Chain Parsers

A nicer definition for *pChainr* is:

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

The Chomsky Hierarchy

The Chomsky hierarchy:

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

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

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

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

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

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

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

$$
\begin{aligned}
&\mathit{times} && :: \mathit{Int} \to \mathit{Parser\ Char\ a} \to \mathit{Parser\ Char\ a} \\
&0\ \text{`}\mathit{times}\text{`}\ p && = \mathit{pure}\ [\,] \\
&n\ \text{`}\mathit{times}\text{`}\ p && = (:) <\$> p <*> (n-1)\ \text{`}\mathit{times}\text{`}\ p \\
&\mathit{abc}\ n && = n <\$\ \ (n\ \text{`}\mathit{times}\text{`}\ a) \\
& && \quad\ <*\ \ (n\ \text{`}\mathit{times}\text{`}\ b) \\
& && \quad\ <*\ \ (n\ \text{`}\mathit{times}\text{`}\ c) \\
&\mathit{ABC} && = \mathit{foldr}\ (<|>)\ \mathit{empty}\ [\mathit{abc}\ n \mid n \leftarrow 0\,..]
\end{aligned}
$$

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

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

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

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

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)]
```

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

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

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

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:

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

Continued ..

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

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

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.

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

- 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.