Universiteit Utrecht

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

Lecture C-10: Parser Combinators - Introduction

USCS 2015

Doaitse Swierstra

Utrecht University

July 6-17, 2015

1. Domain Specific Languages

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

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
- 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.

New idea:

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

2. Elementary Parser Combinators

2.1 What are parser combinators

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.

2.2 Elementary Combinators

Elementary Parsers

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

	Haskell
alternative	$p \ \langle\vert\rangle \ q$
composition	$p \ \langle * \rangle \ q$
terminals	*pSym* `'s'`
empty string	*pSucceed*

Building Parsers out of Parsers

Other operators, e.g. from EBNF:

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

The Type of a Parser

- A parser takes a string and returns, as a proof of its success not an abstract syntax tree, but a result of type a

  > $String \rightarrow a$

- A parser may not consume the whole input, so we return the unused part of the input:

  > $String \rightarrow (a, String)$

- There may be many possible ways of recognizing a value of type a at the beginning of the input:

  > $String \rightarrow [(a, String)]$

- Why should we only accept character strings? Everything goes:

  > $[s] \rightarrow [(a, [s])]$

Types of the Elementary Combinators

$$\textbf{type}\ Parser\ s\ a = [s] \to [(a, [s])]$$

Types

$$
\begin{array}{lll}
\langle|\rangle & :: Parser\ s\ a & \to Parser\ s\ a \to Parser\ s\ a \\
\langle*\rangle & :: Parser\ s\ a & \to Parser\ s\ b \to Parser\ s\ (a, b) \\
\langle*\rangle & :: Parser\ s\ (b \to a) & \to Parser\ s\ b \to Parser\ s\ a \\
pSym & :: Eq\ s \Rightarrow s & \to Parser\ s\ s \\
pSucceed & :: a & \to Parser\ s\ a \\
pFail & :: & Parser\ s\ a
\end{array}
$$

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

2.3 The “List of Successes” Implementation

The "List of Successes" Implementation

$$
\begin{array}{lll}
pFail\ inp & = & [\,] \\
pSucceed\ v\ inp & = & [(v, inp)] \\
(p \langle|\rangle\ q)\ inp & = & p\ inp + q\ inp \\
pSym\ a\ (s : ss) \mid a \equiv s & = & [(s, ss)] \text{ -- !!} Eq\ s \\
pSym\ a\ \_ & = & [\,] \\
(p \langle * \rangle\ q)\ inp & = & [(b2a\ b, rr) \mid (b2a, r) \leftarrow p\ inp \\
 & & \ , (b, \ rr) \leftarrow q\ r \\
 & &]
\end{array}
$$

$$pAB = ((pSucceed\ (,)\ \langle * \rangle\ pSym\ \texttt{'A'})\ \langle * \rangle\ pSym\ \texttt{'B'})$$

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:

$$pAB = \quad pSucceed\ (,)\ \langle * \rangle\ pSym\ \texttt{'A'}\ \ \langle * \rangle\ pSym\ \texttt{'B'}$$

Capturing the essence of $\langle * \rangle$

Suppose we want to deal with possibly failing computations 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* $(b \rightarrow a)$ 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 $\langle * \rangle$ (Cont)

We capture this pattern as follows:

$$
\begin{array}{l}
\textbf{class } \mathit{Applicative}\ p \textbf{ where} \\
\quad (\ \langle * \rangle\ ) :: p\ (b \to a) \to p\ b \to p\ a \\
\quad \mathit{pure} \quad :: a \qquad\qquad\qquad \to p\ a \\
\quad (\ \langle \$ \rangle\ ) :: (b \to a) \quad \to p\ b \to p\ a \\
\quad f\ \langle \$ \rangle\ p = \mathit{pure}\ f\ \langle * \rangle\ p \\
\quad \ldots \\
\textbf{instance } \mathit{Applicative}\ \mathit{Maybe} \textbf{ where} \\
\quad \mathit{Just}\ f\ \langle * \rangle\ \mathit{Just}\ v = \mathit{Just}\ (f\ v) \\
\quad \_ \qquad\ \langle * \rangle\ \_ \qquad = \mathit{Nothing} \\
\quad \mathit{pure} \quad v \qquad\quad = \mathit{Just}\ v
\end{array}
$$

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 build a p-value labelled with type a.

Aside

Using the Haskell class system and its extensions we can denote the application also as:

$$iI\ f\ a_1\ a_2\ a_3\ Ii$$

The symbosls iI and Ii are so-called Idiom brackets.

All *Monads* are *Applicative*

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

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Applicative}\ m\ \textbf{where} \\
&\quad f \langle * \rangle v = \textbf{do}\ \mathit{ff} \leftarrow f \\
&\qquad\qquad\qquad\quad vv \leftarrow v \quad \text{-- no reference to fv in rhs} \\
&\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{ff}\ vv) \\
&\quad \mathit{pure} \quad = \mathit{return}
\end{aligned}
$$

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*:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Alternative}\ m\ \textbf{where} \\
&\quad (\langle|\rangle) :: m\ a \to m\ a \to m\ a \\
&\quad \mathit{empty} :: \qquad\qquad\quad m\ a \\
&\textbf{instance}\ \mathit{Alternative}\ \mathit{Maybe}\ \textbf{where} \\
&\quad \mathit{Just}\ l \ \langle|\rangle\ \_ = \mathit{Just}\ l \\
&\quad \_ \qquad \langle|\rangle\ r = r \\
&\quad \mathit{empty} \qquad\ \ = \mathit{Nothing}
\end{aligned}
$$

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!

2.4 Developing an Embedded Domain Specific Language

Useful functions I

Because the pattern:

$$pSucceed\, f \;\langle * \rangle\; p$$

occurs so often we define

$\langle \$ \rangle$

$$f \;\langle \$ \rangle\; p = pSucceed\, f \;\langle * \rangle\; p$$

so we can write the previous function as:

$$pAB = (,) \;\langle \$ \rangle\; pSym\ \texttt{'A'} \;\langle * \rangle\; pSym\ \texttt{'B'}$$

Useful functions II

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

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

We define

$$
\begin{aligned}
&p\ \langle*\ q = (\lambda x\ \_ \rightarrow x)\ \langle\$\rangle\ p\ \langle*\rangle\ q \\
&p\ *\rangle\ q = (\lambda\_\ y \rightarrow y)\ \langle\$\rangle\ p\ \langle*\rangle\ q \\
&f\ \langle\$\ q = \mathit{pSucceed}\ f\ \langle*\ q
\end{aligned}
$$

So we can now write:

$$
\begin{aligned}
&\mathit{pIfStat} = \mathit{semIfStat}\ \langle\$\ \mathit{pIfToken} \quad \langle*\rangle\ \mathit{pExpr} \\
&\qquad \langle*\ \mathit{pThenToken}\ \langle*\rangle\ \mathit{pExpr}
\end{aligned}
$$

EBNF extensions

$$\mathbf{infixl}\ 2\ \mathit{opt}$$

$$\mathit{opt} :: \mathit{Parser}\ s\ a \to a \to \mathit{Parser}\ s\ a$$

$$p\ \text{`}\mathit{opt}\text{`}\ v = p\ \langle|\rangle\ \mathit{pSucceed}\ v$$

$$\mathit{pList} :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ [a]$$

$$\mathit{pList}\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ \mathit{pList}\ p\ \text{`}\mathit{opt}\text{`}\ [\,]$$

In the library we have special greedy versions which chooses 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:

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

$$\begin{aligned}
pP = {} & (max \cdot (+1)) \langle\$\; pSym \text{ '('} \;\langle*\rangle\; pP \;\langle*\; pSym \text{ ')'} \\
& \qquad\qquad\qquad\qquad \langle*\rangle\; pP \\
& \quad \text{`}opt\text{`} \\
& \quad 0
\end{aligned}$$

Parsing Expressions with Left-Associative Operators

§2.4

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

Left Factorisation

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

$$\begin{aligned} p = &\quad f \langle\$\rangle\ q \langle*\rangle\ r1 \\ &\langle|\rangle\ g \langle\$\rangle\ q \langle*\rangle\ r2 \end{aligned}$$

So we define:

$$p \langle{*}{*}\rangle\ q :: Parser\ s\ b \to Parser\ s\ (b \to a) \to Parser\ s\ a$$
$$p \langle{*}{*}\rangle\ q = (\lambda pv\ qv \to qv\ pv)\ \langle\$\rangle\ p\ \langle*\rangle\ q$$

So we can replace the above code by:

$$p = q \langle{*}{*}\rangle\ (flip\ f\ \langle\$\rangle\ r1\ \langle|\rangle\ flip\ g\ \langle\$\rangle\ r2)$$
$$flip\ f\ x\ y = f\ y\ x$$

More Useful Combinators

$\langle ?? \rangle$

$$p \;\langle ?? \rangle\; q \;::\; Parser\ s\ a \rightarrow Parser\ s\ (a \rightarrow a) \rightarrow Parser\ s\ a$$
$$p \;\langle ?? \rangle\; q = p \;\langle ** \rangle\; (q\ `opt`\ id)$$

pChainr

$$pChainr\ sep\ p = p \;\langle ?? \rangle\; (flip \;\langle \$ \rangle\; sep \;\langle * \rangle\; pChainr\ sep\ p)$$

pParens

$$pParens \;::\; s \rightarrow s \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a$$
$$pParens\ l\ r\ p = pSym\ l \;*\rangle\; p \;\langle *\; pSym\ r$$

Left Factorisation II

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

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

$$\begin{array}{ll} \mathit{pFactor} & = \mathit{Lambda} \langle\$\ \mathit{pSym}\ \texttt{'\textbackslash\textbackslash'}\ \langle*\rangle\ \mathit{pIdent} \\ & \qquad\qquad \langle*\ \mathit{pSym}\ \texttt{'.'}\ \langle*\rangle\ \mathit{pExpr} \\ & \quad \langle|\rangle\ \mathit{pParens}\ \texttt{'('}\ \texttt{')'}\ \mathit{pExpr} \\ & \quad \langle|\rangle\ \mathit{Ident}\ \langle\$\rangle\ \mathit{pIdent} \\ \mathit{pExpr} & = \mathit{pChainl}\ (\mathit{pSucceed}\ \mathit{App}) \\ & \quad (\mathit{pFactor}\ \langle??\rangle\ (\mathit{TypedExpr}\ \langle\$\ \mathit{pTok}\ \texttt{"::"} \\ & \qquad\qquad\qquad\qquad \langle*\rangle\ \mathit{pTypeDescr})) \end{array}$$

2.5 Monadic Parsers

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

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

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

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

$$
\begin{array}{lll}
\textbf{instance}\ \mathit{Monad}\ (\mathit{Parser}\ s)\ \textbf{where} & & \\
\quad p\ (\gg\!=)\ q & = \lambda inp \rightarrow [\,(qv, rr) \mid (pv, r) \leftarrow p\ \ inp & \\
 & \qquad\qquad\qquad\quad\ ,\ (qv, rr) \leftarrow q\ pv\ r & \\
 & \qquad\qquad] & \\
\quad return\ v & = \lambda inp \rightarrow [\,(v, inp)\,] & \\
as & :: \mathit{Parser}\ \mathit{Char}\ \mathit{Int} & \\
as & = length\ \langle\$\ pList\ (pSym\ \texttt{'a'}) & \\
bc\ n & = n\ \langle\$\ (n\ \text{`}times\text{`}\ b)\ \langle*\ (n\ \text{`}times\text{`}\ c) & \\
ABC & = \textbf{do}\ n \leftarrow as & \\
 & \qquad\ \ bc\ n &
\end{array}
$$

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:

$$
\begin{aligned}
&\textbf{type } Parser\ s\ a = [s] \rightarrow [(a, [s], Int)] \\
&(p \circledast q)\ inp \quad = [(pv\ qv, rr, pc + qc) \mid (pv, r, pc) \leftarrow p\ inp \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , (qv, rr, qc) \leftarrow q\ r \\
&\qquad\qquad\qquad\quad]
\end{aligned}
$$

...

Continued ..

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

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

2.6 Problems

Problems with Erroneous Input

- If your input does not conform to the language recognised 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

This problem can be partially been cured by using chaining combinators.

Problems with Space Consumption

Our naïve "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.