Defining Parser Combinators using Attribute Grammars

L. Thomas van Binsbergen

Royal Holloway, University of London

11 May, 2015

Designing Parser Combinators using AGs

- Showcasing Functional Programming:
  1. Parser combinators are a hot topic in FP.
  2. Their definitions show many features of FP.
- Why use Attribute Grammars?
  1. AGs allow me to explain definitions without code.
  2. Parser combinators form a test case of a more general idea.
- Following this general idea I have implemented:
  1. The parser combinators presented today.
  2. A small imperative programming language (ETAPS demo).
  3. Parser combinators for generalised top-down (GLL) parsing.

Designing Combinator Languages using AGs

- Showcasing Functional Programming:
  1. Parser combinators are a hot topic in FP.
  2. Their definitions show many features of FP.
- Why use Attribute Grammars?
  1. AGs allow me to explain definitions without code.
  2. Parser combinators form a test case of a more general idea.
- Following this general idea I have implemented:
  1. The parser combinators presented today.
  2. A small imperative programming language (ETAPS demo).
  3. Parser combinators for generalised top-down (GLL) parsing.

Designing EDSLs using AGs

- Showcasing Functional Programming:
  1. Parser combinators are a hot topic in FP.
  2. Their definitions show many features of FP.
- Why use Attribute Grammars?
  1. AGs allow me to explain definitions without code.
  2. Parser combinators form a test case of a more general idea.
- Following this general idea I have implemented:
  1. The parser combinators presented today.
  2. A small imperative programming language (ETAPS demo).
  3. Parser combinators for generalised top-down (GLL) parsing.

Introduction

Parser Combinators

- A parser combinator library is like an axiomatic system.
- Simple parsers are combined to derive more complex parsers.
- Benefits: 1) **compositional**, 2) **type-safe** semantic actions.

Attribute Grammars

- AGs are used to describe programming language semantics.
- Very high level: programming by thinking of trees.
- Benefits:
  1) **concern-separation** 2) **aspect-oriented** 3) modular.

Example

BNF grammar

$$X ::= \texttt{'a'}\ \texttt{'b'}\ X \mid \epsilon$$

Combinator expression

$$pX = \textbf{char}\ \texttt{'a'}\ \langle * \rangle\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ pX\ \langle | \rangle\ \epsilon$$

How many occurrences of "ab" are there in a string of "ab"s?

$$pX = (1+)\ \langle \$\ \textbf{char}\ \texttt{'a'}\ \langle *\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ pX\ \langle | \rangle\ 0\ \langle \$\ \epsilon$$

Example

Running the parser

$$\mathit{parse}\ \mathit{pX}\ \texttt{"ababab"} = [3] \quad \text{-- singleton list with 3 in it}$$

Combinator expression

$$\mathit{pX} = \textbf{char}\ \texttt{'a'}\ \langle * \rangle\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ \mathit{pX}\ \langle | \rangle\ \epsilon$$

How many occurrences of "ab" are there in a string of "ab"s?

$$\mathit{pX} = (1+)\ \langle \$\ \textbf{char}\ \texttt{'a'}\ \langle *\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ \mathit{pX}\ \langle | \rangle\ 0\ \langle \$\ \epsilon$$

Example

Running the parser

$parse\ pX$ `"ababab"` $= [3]$ -- singleton list with 3 in it

Running the parser (2)

$parse\ pX$ `"abababcdef"` $= [\,]$ -- no parse

How many occurrences of "ab" are there in a string of "ab"s?

$pX = (1+) \langle\$\ \textbf{char}\ \texttt{'a'}\ \langle*\ \textbf{char}\ \texttt{'b'}\ \langle*\rangle\ pX\ \langle|\rangle\ 0\ \langle\$\ \epsilon$

Parser Combinators as Domain Specific Language (DSL)

Valid documents of our DSL

$$
\begin{array}{lcl}
\textit{Document} & ::= & \textit{Rule}^{*} \\
\textit{Rule} & ::= & \textit{Identifier}\ \texttt{'='}\ \textit{Cexpr} \\
\textit{Cexpr} & ::= & \epsilon \\
 & | & \textbf{char}\ \textit{Char} \\
 & | & \textit{Cexpr}\ \langle | \rangle\ \textit{Cexpr} \\
 & | & \textit{Cexpr}\ \langle * \rangle\ \textit{Cexpr} \\
 & | & \textit{Identifier}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{Char} & ::= & \texttt{'a'} \\
 & | & \texttt{'b'} \\
 & | & \ldots
\end{array}
$$

Parser Combinators as Embedded DSL (EDSL)

Grammar of Combinator Expressions

$$
\begin{aligned}
\mathit{Cexpr} ::= {} & \epsilon \\
\mid {} & \textbf{char}\ \mathit{Char} \\
\mid {} & \mathit{Cexpr}\ \langle|\rangle\ \mathit{Cexpr} \\
\mid {} & \mathit{Cexpr}\ \langle*\rangle\ \mathit{Cexpr}
\end{aligned}
$$

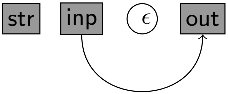
Defining EDSL semantics using Attribute Grammars

Attributes

attr *Cexpr*
  inh *str* :: *String*
  inh *inp* :: *Int*
  syn *out* :: [*Int*]   -- list of integers

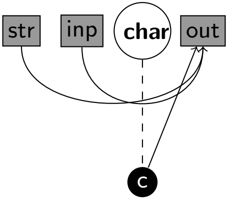
- Every combinator expressions answers the question: "Which substrings of *str* (starting at *inp*) do you recognise?"

Epsilon



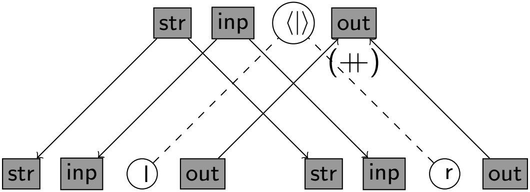
- Substrings: $[str_{inp,inp}]$

Character

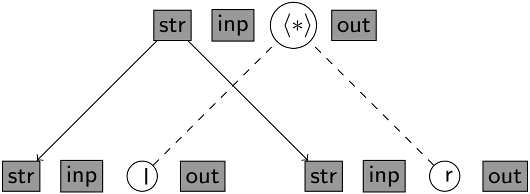


- Substrings: $[str_{inp,inp+1}]$ or [].

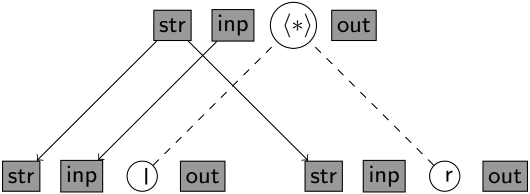
Alternatives



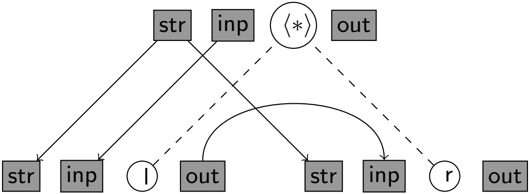
Sequencing (1)



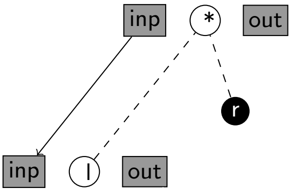
Sequencing (1)



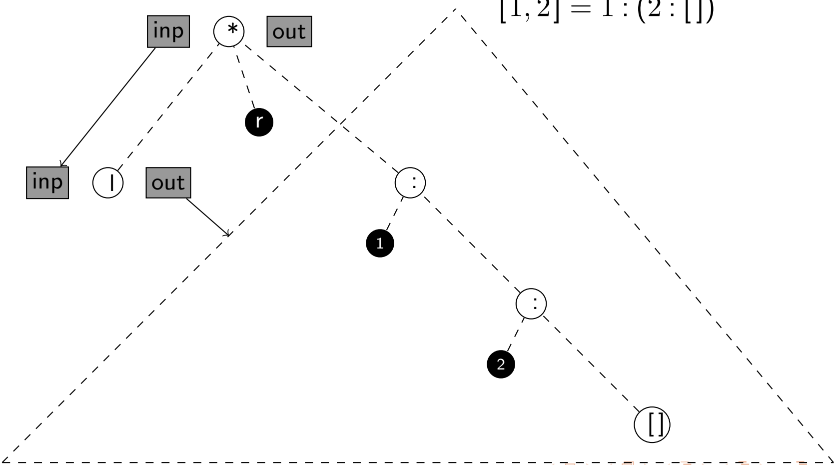
Sequencing (1)



Sequencing (2)

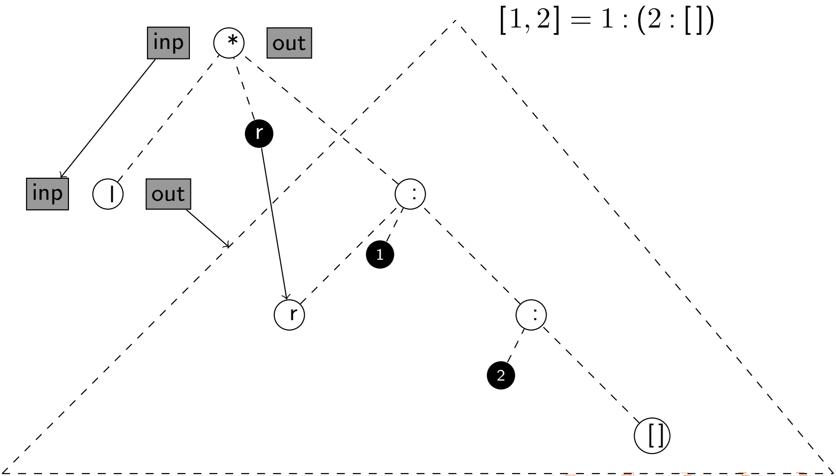


Sequencing (2)

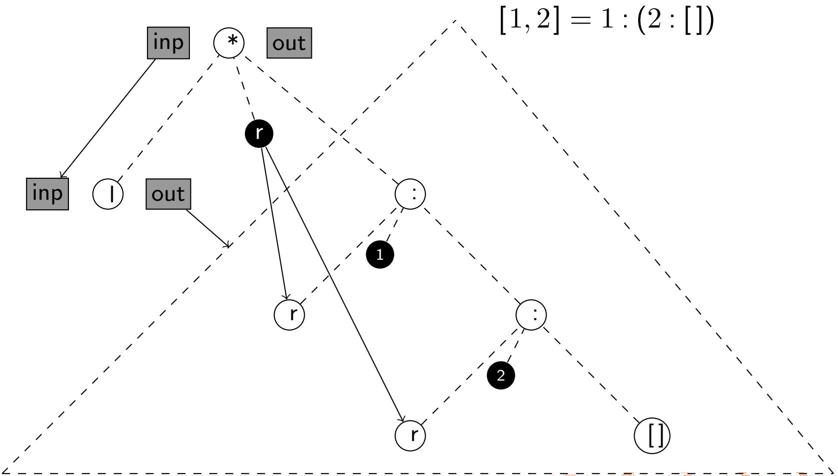


$$[1, 2] = 1 : (2 : [])$$

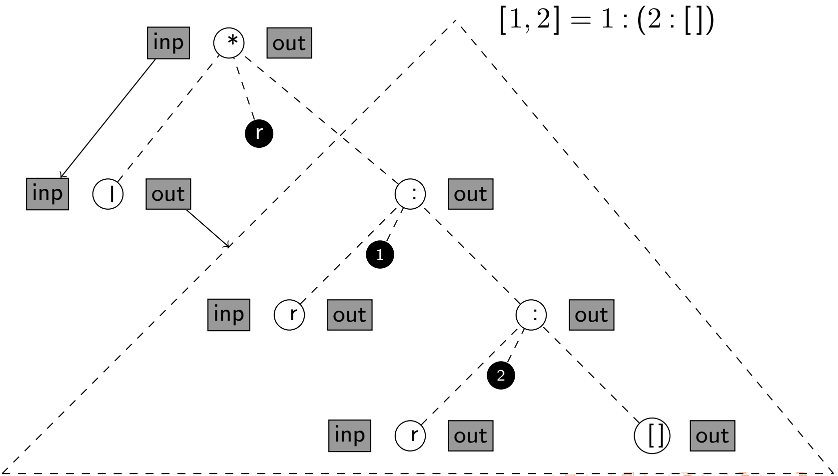
Sequencing (2)



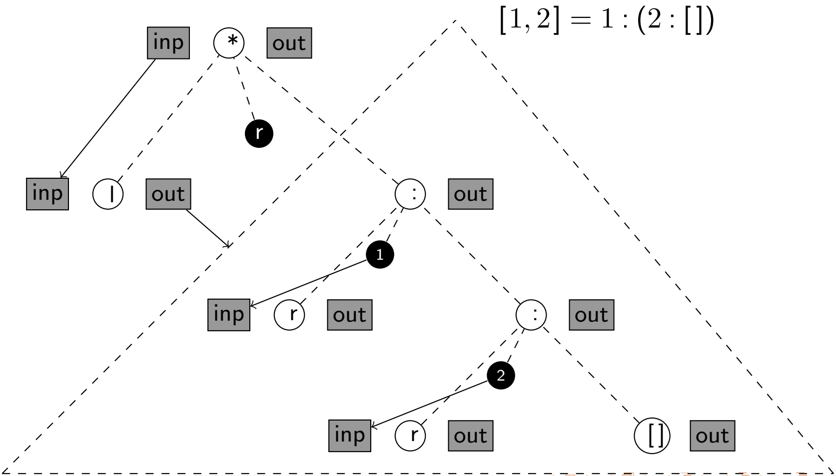
Sequencing (2)



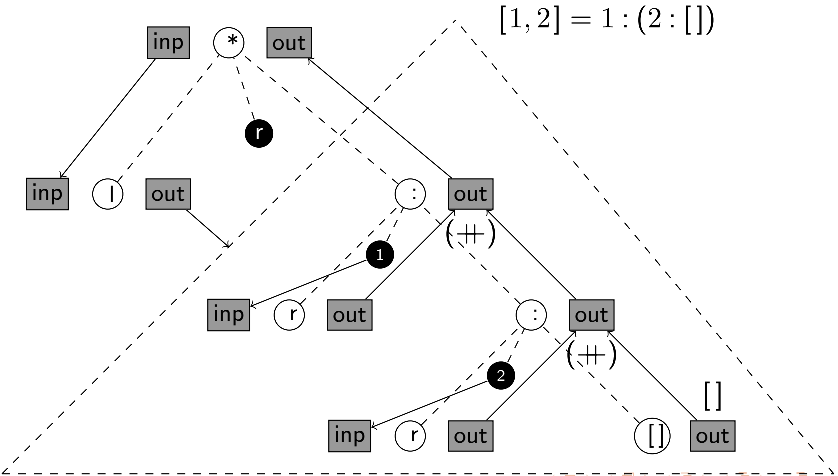
Sequencing (2)



Sequencing (2)



Sequencing (2)



Example

BNF grammar

$$X ::= \texttt{'a'}\ \texttt{'b'}\ X \mid \epsilon$$

Progress so far

$$pX = \textbf{char}\ \texttt{'a'}\ \langle * \rangle\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ pX\ \langle | \rangle\ \epsilon$$

Up next

$$pX = (1+)\ \langle \$\ \textbf{char}\ \texttt{'a'}\ \langle *\ \textbf{char}\ \texttt{'b'}\ \langle * \rangle\ pX\ \langle | \rangle\ 0\ \langle \$\ \epsilon$$

Generalised Algebraic DataType (GADT)

Combinator Expressions, with semantic results

data *Cexpr x* **where**

$$
\begin{array}{lll}
\textbf{char} & :: Char & \rightarrow Cexpr\ Char \\
\textbf{satisfy} & :: a & \rightarrow Cexpr\ a \\
\langle|\rangle & :: Cexpr\ a \rightarrow Cexpr\ a & \rightarrow Cexpr\ a \\
\langle*\rangle & :: Cexpr\ (b \rightarrow a) \rightarrow Cexpr\ b & \rightarrow Cexpr\ a
\end{array}
$$

- *Cexpr x* builds parsers that produce *x*'s.
- Piggybacking on Haskell's type system.

Combinators for applying semantic actions

- $\langle * \rangle :: Cexpr\ (b \to a) \to Cexpr\ b \to Cexpr\ a$
- **satisfy** $:: a \to Cexpr\ a$
- $const :: a \to (b \to a)$
  $const\ x\ y = x$

Definitions

- $\langle \$ \rangle :: (b \to a) \to Cexpr\ b \to Cexpr\ a$
- $\langle \$ :: a \to Cexpr\ b \to Cexpr\ a$
- $\langle * :: Cexpr\ a \to Cexpr\ b \to Cexpr\ a$

Combinators for applying semantic actions

- $\langle * \rangle :: Cexpr\ (b \to a) \to Cexpr\ b \to Cexpr\ a$
- $\mathbf{satisfy} :: a \to Cexpr\ a$
- $const :: a \to (b \to a)$
  $const\ x\ y = x$

Definitions

- $\langle \$ \rangle :: (b \to a) \to Cexpr\ b \to Cexpr\ a$
  $f\ \langle \$ \rangle\ p = \mathbf{satisfy}\ f\ \langle * \rangle\ p$
- $\langle \$ :: a \to Cexpr\ b \to Cexpr\ a$
- $\langle * :: Cexpr\ a \to Cexpr\ b \to Cexpr\ a$

Combinators for applying semantic actions

- $\langle * \rangle$ :: $Cexpr\ (b \to a) \to Cexpr\ b \to Cexpr\ a$
- **satisfy** :: $a \to Cexpr\ a$
- $const$ :: $a \to (b \to a)$
  $const\ x\ y = x$

Definitions

- $\langle \$ \rangle$ :: $(b \to a) \to Cexpr\ b \to Cexpr\ a$
  $f\ \langle \$ \rangle\ p =$ **satisfy** $f\ \langle * \rangle\ p$
- $\langle \$$ :: $a \to Cexpr\ b \to Cexpr\ a$
  $f\ \langle \$\ p = const\ f\ \langle \$ \rangle\ p$
- $\langle *$ :: $Cexpr\ a \to Cexpr\ b \to Cexpr\ a$

Combinators for applying semantic actions

- $\langle * \rangle :: Cexpr\ (b \rightarrow a) \rightarrow Cexpr\ b \rightarrow Cexpr\ a$
- $\textbf{satisfy} :: a \rightarrow Cexpr\ a$
- $const :: a \rightarrow (b \rightarrow a)$
  $const\ x\ y = x$

Definitions

- $\langle \$ \rangle :: (b \rightarrow a) \rightarrow Cexpr\ b \rightarrow Cexpr\ a$
  $f\ \langle \$ \rangle\ p = \textbf{satisfy}\ f\ \langle * \rangle\ p$
- $\langle \$ :: a \rightarrow Cexpr\ b \rightarrow Cexpr\ a$
  $f\ \langle \$\ p = const\ f\ \langle \$ \rangle\ p$
- $\langle * :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ a$
  $l\ \langle *\ r = const\ \langle \$ \rangle\ l\ \langle * \rangle\ r$

Derived Combinators

Extended BNF (EBNF)

- $optional :: Cexpr\ a \rightarrow Cexpr\ (Maybe\ a)$
- $many :: Cexpr\ a \rightarrow Cexpr\ [a]$
- $some :: Cexpr\ a \rightarrow Cexpr\ [a]$
- $sepBy :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ [a]$

Derived Combinators

Extended BNF (EBNF)

- $optional :: Cexpr\ a \rightarrow Cexpr\ (Maybe\ a)$
  $optional\ p = Just\ \langle\$\rangle\ p\ \langle|\rangle\ \textbf{satisfy}\ Nothing$
- $many :: Cexpr\ a \rightarrow Cexpr\ [a]$
- $some :: Cexpr\ a \rightarrow Cexpr\ [a]$
- $sepBy :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ [a]$

Derived Combinators

Extended BNF (EBNF)

- $optional :: Cexpr\ a \rightarrow Cexpr\ (Maybe\ a)$
  $optional\ p = Just\ \langle\$\rangle\ p\ \langle|\rangle\ \mathbf{satisfy}\ Nothing$
- $many :: Cexpr\ a \rightarrow Cexpr\ [a]$
  $many\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ p\ \langle|\rangle\ \mathbf{satisfy}\ [\,]$
- $some :: Cexpr\ a \rightarrow Cexpr\ [a]$
- $sepBy :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ [a]$

Derived Combinators

Extended BNF (EBNF)

- $optional :: Cexpr\ a \rightarrow Cexpr\ (Maybe\ a)$
  $optional\ p = Just\ \langle\$\rangle\ p\ \langle|\rangle\ \mathbf{satisfy}\ Nothing$
- $many :: Cexpr\ a \rightarrow Cexpr\ [a]$
  $many\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ p\ \langle|\rangle\ \mathbf{satisfy}\ []$
- $some :: Cexpr\ a \rightarrow Cexpr\ [a]$
  $some\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ p$
- $sepBy :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ [a]$

Derived Combinators

Extended BNF (EBNF)

- $optional :: Cexpr\ a \rightarrow Cexpr\ (Maybe\ a)$
  $optional\ p = Just\ \langle\$\rangle\ p\ \langle|\rangle\ \textbf{satisfy}\ Nothing$
- $many :: Cexpr\ a \rightarrow Cexpr\ [a]$
  $many\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ p\ \langle|\rangle\ \textbf{satisfy}\ [\,]$
- $some :: Cexpr\ a \rightarrow Cexpr\ [a]$
  $some\ p = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ p$
- $sepBy :: Cexpr\ a \rightarrow Cexpr\ b \rightarrow Cexpr\ [a]$
  $sepBy\ p\ sep = (:)\ \langle\$\rangle\ p\ \langle*\rangle\ many\ (sep *\rangle\ p)$

Derived Combinators (2)

Others

- $within :: Cexpr\ b \rightarrow Cexpr\ a \rightarrow Cexpr\ c \rightarrow Cexpr\ a$
  $within\ l\ p\ r = l\ {*}\rangle\ p\ \langle{*}\ r$
- $parenthesised :: Cexpr\ a \rightarrow Cexpr\ a$
  $parenthesised\ p = within\ (\mathbf{char}\ \text{'('})\ p\ (\mathbf{char}\ \text{')'})$

Take home message

- Parser Combinators are very expressive.
- Users can add their own extensions to BNF.
- Semantic actions are type-checked.
- Easily implemented in a functional programming language.