Generalised Parsing and Combinator Parsing

A Happy Marriage?

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org

Royal Holloway, University of London

October 30, 2016

1. Conventional Parser Combinators
   - Recursive Descent
   - Monadic Parser Combinators
   - Benefits of Conventional Parser Combinators
2. Aspects of Generalised Parsing
   - Observable Sharing
   - Derivation Representation
3. Combinators for Generalised Parsing
   - Design Space Exploration
   - Challenges

Section 1

Conventional Parser Combinators

Recursive Descent Parsing

- Every symbol is implemented by a *parse function*
- A parse function:
  - Receives the input string, and a *pivot*
  - Returns a new pivot, and a bit of parse tree / semantic value
- The parse function for a nonterminal:
  - *Chooses* one of its productions (**somehow**)
  - Executes the symbols of the production in *sequence*

Combinator Approach

- Forget all about symbols and production
- Focus on parse functions and their composition!

$$\textbf{type}\ Parser\ t\ a = [t] \rightarrow Int \rightarrow [(Int, a)]$$

- Full power of host language available to construct parsers

$$myparser :: \ldots \rightarrow Parser\ t\ a$$

Elementary matchers - create parsers from non-parser values

$$\begin{aligned} term &:: t &&\rightarrow Parser\ t\ t \\ pred &:: (t \rightarrow Bool) &&\rightarrow Parser\ t\ t \\ prefix &:: [t] &&\rightarrow Parser\ t\ [t] \end{aligned}$$

$$\textbf{type}\ Parser\ t\ a = [t] \rightarrow Int \rightarrow [(Int, a)]$$

- Full power of host language available to construct parsers

$$myparser :: ... \rightarrow Parser\ t\ a$$

Elementary combinators - are defined by library internals

- e.g. *seq* for placing parsers in sequence
- e.g. *alt* for choosing between parsers

$$alt :: Parser\ t\ a \rightarrow Parser\ t\ a \rightarrow Parser\ t\ a$$
$$alt\ p\ q\ str\ k = p\ str\ k +\!\!+\ q\ str\ k$$

- Define *return* and $\ggg=$
- Prove monadic laws

Parsers are monadic!

$$return :: a \rightarrow Parser\ t\ a$$
$$return\ a\ str\ l = \ldots$$

$$(\ggg=) :: Parser\ t\ a \rightarrow (a \rightarrow Parser\ t\ b) \rightarrow Parser\ t\ b$$
$$(p \ggg= a2q)\ str\ l = \ \ldots$$

- Define *return* and $\gg\!=$
- Prove monadic laws

Parsers are monadic!

$$return :: a \to Parser\ t\ a$$
$$return\ a\ str\ l = [(l, a)]$$

$$(\gg\!=) :: Parser\ t\ a \to (a \to Parser\ t\ b) \to Parser\ t\ b$$
$$\begin{aligned}(p \gg\!= a2q)\ str\ l = &\ [\ (r, b) \\ &\ \mid (k, a) \leftarrow p\ str\ l \\ &\ ,\ (r, b) \leftarrow (a2q\ a)\ str\ k]\end{aligned}$$

What do combinators offer beyond parser generators?

- Easy to define alternative *elementary matchers/combinators*
- Easy to define *derived combinators*
- Some form of context-sensitivity

- Advantages follow from the simplicity of the underlying algorithm
- There may be *other advantages specific to your application*

Section 2

Aspects of Generalised Parsing

- Conventional P.C. do not provide *any* grammar information

- Generalised Parsing requires *explicit grammar information*, for: *GSS construction, memoisation, curtailing left-recursive calls*...

- At the least, G.P. requires unique identifiers for symbols

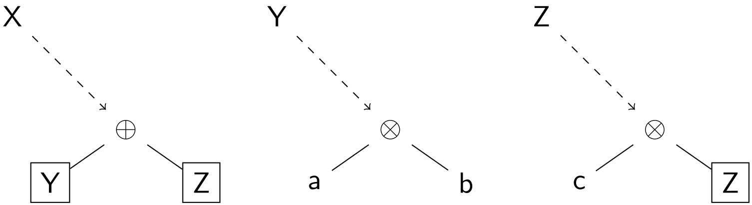
- At the **very** least, G.P. requires identifiers for recursive positions

- All we need is...

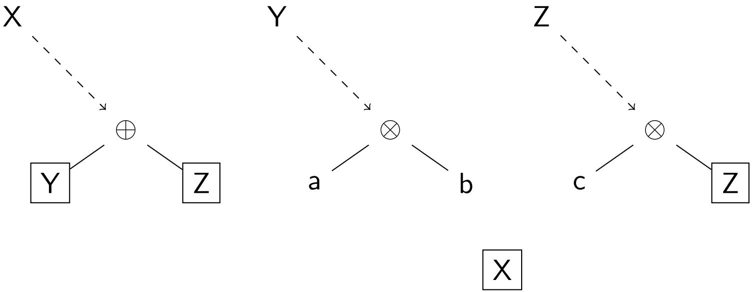
- Conventional P.C. do not provide *any* grammar information
- Generalised Parsing requires *explicit grammar information*, for: *GSS construction, memoisation, curtailing left-recursive calls*...
- At the least, G.P. requires unique identifiers for symbols
- At the **very** least, G.P. requires identifiers for recursive positions
- All we need is... **observable sharing**

Observable sharing

- Detecting that two expressions arose from the same binding
- Simple pure 'solution': Ask the programmer!
- The way we obtain observable sharing influences how derived combinators are defined

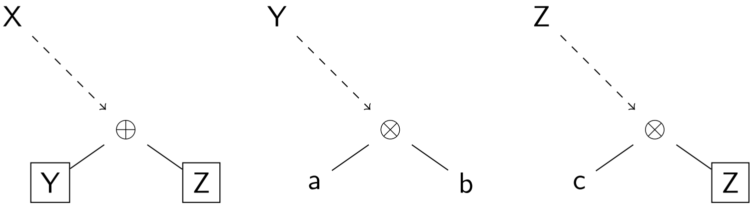


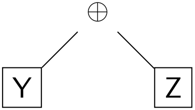
$$X ::= Y \mid Z$$
$$Y ::= `a` \; `b`$$
$$Z ::= `c` \; Z$$



$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{`}\ \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{`}\ Z$$

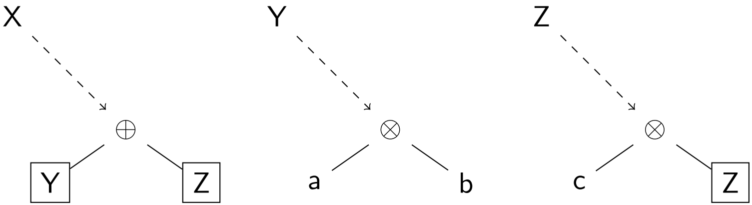
$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$



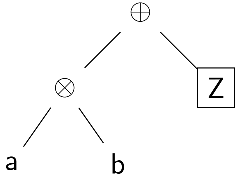


$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{`}\ \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{`}\ Z$$

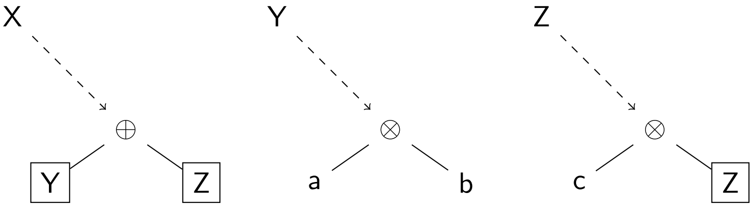
$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$

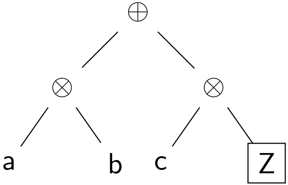


$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{`}\ \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{`}\ Z$$



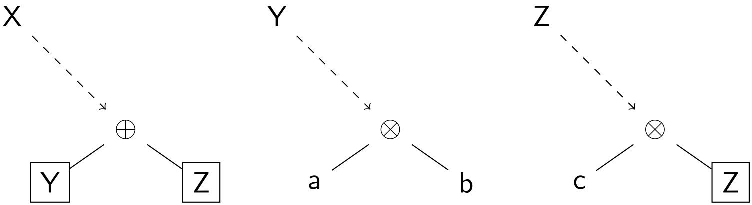
$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$





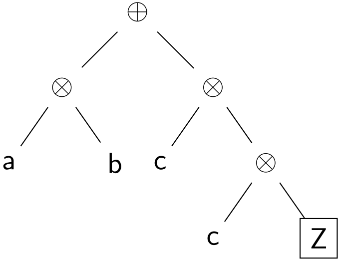
$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{`}\ \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{`}\ Z$$

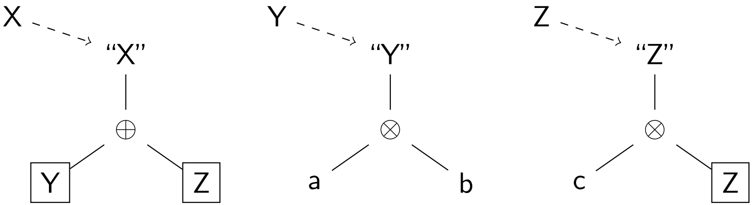
$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$



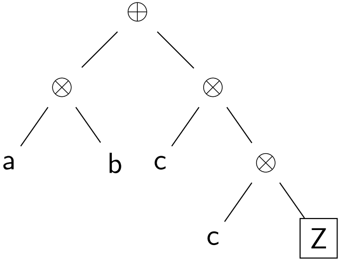
$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{`}\ \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{`}\ Z$$

$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$

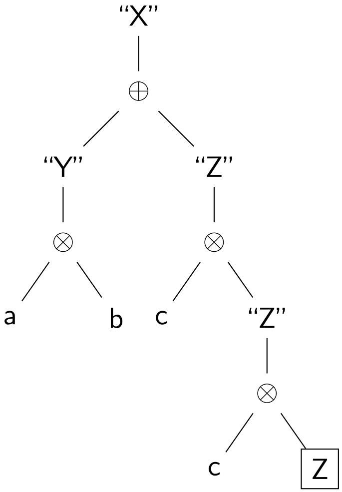




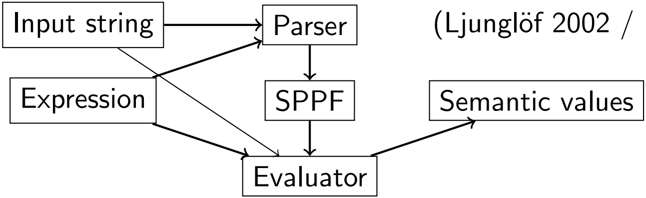
$$X ::= Y \mid Z$$
$$Y ::= \text{`}a\text{` } \text{`}b\text{`}$$
$$Z ::= \text{`}c\text{` } Z$$



$$\text{term} = 1$$
$$\oplus = (+)$$
$$\otimes = (+)$$

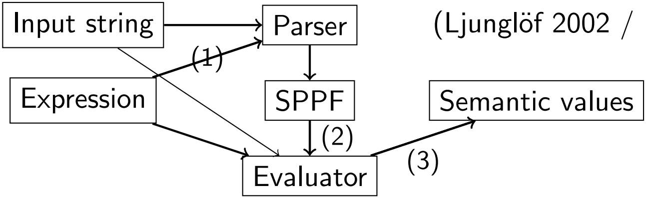

"X"
⊕
"Y"
"Z"
⊗
⊗
a
b
c
"Z"
⊗
c
Z

- A Generalised Parser produces a **representation** of *all* derivations
- Potentially exponentially many derivations may be embedded
- Downstream enumeration would result in exponential runtimes...



(Ljunglöf 2002 / Ridge 2014)

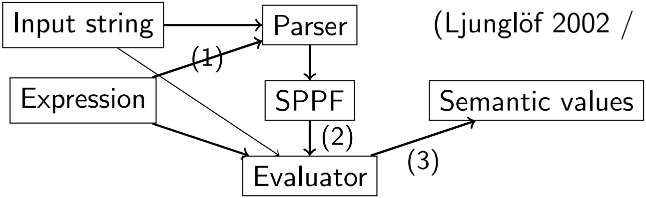
- A Generalised Parser produces a **representation** of *all* derivations
- Potentially exponentially many derivations may be embedded
- Downstream enumeration would result in exponential runtimes...



(Ljunglöf 2002 / Ridge 2014)

How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)

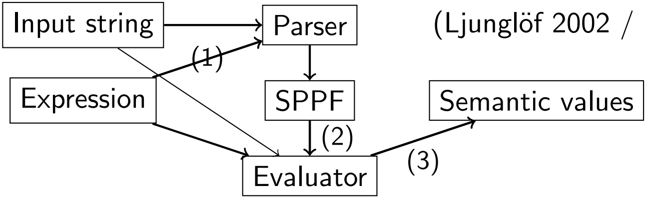


(Ljunglöf 2002 / Ridge 2014)

How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)

- How to provide *expressive*, but *opaque*, strategies?



(Ljunglöf 2002 / Ridge 2014)

How may disambiguation strategies manifest themselves?

1. A more deterministic parsers being constructed
2. Pruning the representation of all derivations (top-down)
3. Choosing between derivations by evaluation (bottom-up)

- How to provide *expressive*, but *opaque*, strategies?
- How do the strategies interact with user-derived combinators?

Section 3

Combinators for Generalised Parsing

Parser Combinators	`parsec`	
	`UU-lib`	
	...	
Explicit Nonterminals	Scheme recognisers	(Johnson 1995)
	`Meerkat`	(Izmaylova/Afroozeh 2015/16)
Grammar Combinators	`P3`	(Ridge 2014)
	GLL.Combinators	(2015/16)
	`grammar-combinators`	(Devriese 2011/12)
Meta-Programming	`BNFC-meta`	(Duregard 2011)
Parser Generators	`Bison`	
	`yacc`	
	`Happy`	
	...	

Properties of a “good” library

- Supports *semantic actions*
- Worst-case runtime $O(n^3)$ & Precomputation at compile-time
- Provides a DSL with little to learn
- Allows users to define derived combinators (with little or no knowledge of internals)

Bonus

- Allows users to define elementary matcher or combinators (without too much effort/risk)
- Supports some form of context-sensitivity (like monadic parsers)

Generalised Parsing and Combinator Parsing

A Happy Marriage?

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org

Royal Holloway, University of London

October 30, 2016

"*To extend your library - How would* **you** *define the following*?"

Alternative Elementary Combinators

- Combinator *pred*, matching terminals satisfying a predicate
- The internal equivalent of Kleene-closure

"*With your library - How can a* **user** *define the following*?"

Derived Combinators

- Combinator *many*, the 'external' Kleene-closure
- Combinator *chainl* for expression grammars

- Ideally knowing nothing or little of the library's internals

"*How do disambiguation strategies mix with derived combinators*?"