Structurally Recursive Descent Parsing

Nils Anders Danielsson (Nottingham)
Joint work with Ulf Norell (Chalmers)

Parser combinators

- Parser combinator libraries are great!
- Elegant code.
- Executable grammars.
- Easy to abstract out recurring patterns.
- Light-weight.
- Nowadays often fast enough.

Simple example

```
expr = _+_ $ term <* sym '+' <*> expr
     |       term
term = ...
```

Simple example

```
expr = _+_ $ expr <* sym '+' <*> term
     |       term
term = ...
```

⊥

Risk of non-termination

- Combinator parsing is not guaranteed to terminate.
- Most combinator parsers fail for left-recursive grammars.
- *Executable* grammars?
- Some errors are not caught at compile-time.

Another problem

- All interesting grammars are cyclic:

```
expr = _+_  $ term  <* sym '+'  <*> expr
     |        term
```

- Cyclic values are hard to understand and reason about.
- How do you implement combinator parsing in a language which requires structural recursion?

Our solution

- Remove cycles by representing grammars as functions from non-terminals to parsers:

```
Grammar tok nt = nt res → Parser tok nt res
```

- Rule out left recursion by restricting the types.

Examples

Example

- Non-terminals:

```
data NT : ParserType where
  expr  : NT _ ℕ
  term : NT _ ℕ
```

- Result type: ℕ.
- Indices ensuring termination: _.
  Inferred automatically.

Example

```
g : Grammar Char NT
g expr  =  _+_   $ ! term  <* sym '+' <*> ! expr
        |           ! term
g term = . . .
```

- Note: *g* is not recursive.
- ! turns a non-terminal into a parser.

Example

```
g : Grammar Char NT
g expr  =  _+_   $ ! term  <* sym '+' <*> ! expr
        |           ! term
g term = . . .
```

- Uses applicative functor interface.
- Monadic interface also possible.

Abstraction

- Much of the flavour of ordinary combinator parsers is preserved.
- Abstraction requires a little work, though.

Abstraction

```
data NT : ParserType where
  lib  : L.Nonterminal NT i r → NT _ r
  expr : NT _ ℕ
  term : NT _ ℕ
  op   : NT _ (ℕ → ℕ → ℕ)
open L.Combinators lib
g : Grammar Char NT
g (lib p) = libraryGrammar p
g expr    = chainl₁ (! term) (! op)
g term    = number | parenthesised (! expr)
g op      = _+_  <$  sym '+'
          | _−_  <$  sym '-'
```

Abstraction

```
data NT : ParserType where
  lib  : L.Nonterminal NT i r → NT _ r
  expr : NT _ ℕ
  term : NT _ ℕ
  op   : NT _ (ℕ → ℕ → ℕ)
open L.Combinators lib
g : Grammar Char NT
g (lib p) = libraryGrammar p
g expr    = chainl₁ (! term) (! op)
g term    = number | parenthesised (! expr)
g op      = _+_  <$  sym '+'
          | _−_  <$  sym '-'
```

Running a parser

```
parse :   Parser tok nt i result
        → Grammar tok nt
        → [tok] → [result × [tok]]
```

How does it work?

Indices

Parsers are indexed on two things:

$Index = Empty \times Corners$

Empty	Does the parser accept the empty string?
Corners	A tree representation of the proper left corners of the parser.

Indices

Empty Does the parser accept the empty string?

$Empty = Bool$

Corners Represents all positions in the grammar in which the parser must not recurse to itself.

data *Corners* : *Set* **where**
  leaf : *Corners*
  step : *Corners* → *Corners*
  node : *Corners* → *Corners* → *Corners*

Some basic combinators

$$\begin{aligned}
\_{<}{*}{>}\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \mathrm{node}\ c_1\ c_2\ \textbf{else}\ c_1) \\
\_|\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \vee e_2, \mathrm{node}\ c_1\ c_2) \\
!\_ &: nt\ (e, c) \to Parser\ (e, \mathrm{step}\ c)
\end{aligned}$$

Some basic combinators

$$\begin{aligned}
\_{<}{*}{>}\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \mathrm{node}\ c_1\ c_2\ \textbf{else}\ c_1) \\
\_|\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \vee e_2, \mathrm{node}\ c_1\ c_2) \\
!\_ &: nt\ (e, c) \to Parser\ (e, \mathrm{step}\ c)
\end{aligned}$$

Some basic combinators

$$\begin{aligned}
\_{<}{*}{>}\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \mathrm{node}\ c_1\ c_2\ \textbf{else}\ c_1) \\
\_|\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \vee e_2, \mathrm{node}\ c_1\ c_2) \\
!\_ &: nt\ (e, c) \to Parser\ (e, \mathrm{step}\ c)
\end{aligned}$$

This does not type check:

$grammar : nt\ (e, c)\ res \to Parser\ tok\ nt\ (e, c)\ res$
$grammar\ \mathrm{rec} = !\,\mathrm{rec}$

Reason: $c \neq \mathrm{step}\ c$.

Some basic combinators

$$\begin{aligned}
\_{<}{*}{>}\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \mathrm{node}\ c_1\ c_2\ \textbf{else}\ c_1) \\
\_|\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \vee e_2, \mathrm{node}\ c_1\ c_2) \\
!\_ &: nt\ (e, c) \to Parser\ (e, \mathrm{step}\ c)
\end{aligned}$$

This works, though:

$grammar : nt\ (e, c)\ res \to Parser\ tok\ nt\ (e, c)\ res$
$grammar\ \mathrm{rec} = sym\ c \ast{>}\ !\,\mathrm{rec}$

Reason: *sym c* must consume a token.

Some basic combinators

$$\begin{aligned}
\_{<}{*}{>}\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \mathrm{node}\ c_1\ c_2\ \textbf{else}\ c_1) \\
\_|\_ &: Parser\ (e_1, c_1) \to Parser\ (e_2, c_2) \to \\
&\quad Parser\ (e_1 \vee e_2, \mathrm{node}\ c_1\ c_2) \\
!\_ &: nt\ (e, c) \to Parser\ (e, \mathrm{step}\ c)
\end{aligned}$$

Indirect left recursion also fails:

$grammar : nt\ (e, c)\ res \to Parser\ tok\ nt\ (e, c)\ res$
$grammar\ \mathrm{rec} \quad = !\,\mathrm{other}\ {<}{*}{>}\ \ldots$
$grammar\ \mathrm{other} = many\ p\ {<}{*}{>}\ !\,\mathrm{rec}$

Indices can be useful anyway

- Parsing zero or more things:

$$\begin{aligned} \mathit{many}\ &:\ \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\mathsf{false}, c)\ r \\ &\to \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \_\qquad\quad [r] \end{aligned}$$

- Note that the input parser must not accept the empty string.
- Even if the backend can handle *many empty* it seems reasonable to assume that it is a bug.

Backend

- Simple backtracking implementation. (So far.)
- Lexicographic structural recursion over:
  1. An upper bound on the length of the input string.
  2. The *Corners* index.
  3. The structure of the parser.

Expressive power?

- Can define grammars with an infinite number of non-terminals:

$$\begin{aligned} &\mathbf{data}\ \mathit{NT} : \mathit{ParserType}\ \mathbf{where} \\ &\quad \mathrm{a}^{1+}\_ : \mathbb{N} \to \mathit{NT}\ \_\ \mathit{Unit} \\ &g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\ &g\ (\mathrm{a}^{1+}\ \mathrm{zero}) \qquad\ = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ \mathit{return}\ \mathrm{unit} \\ &g\ (\mathrm{a}^{1+}\ (\mathrm{suc}\ n)) = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ !\,(\mathrm{a}^{1+}\ n) \end{aligned}$$

- Can use this to define non-context-free languages: $a^n b^n c^n$.

Expressive power?

- Can define grammars with an infinite number of non-terminals:

$$\begin{aligned} &\mathbf{data}\ \mathit{NT} : \mathit{ParserType}\ \mathbf{where} \\ &\quad \mathrm{a}^{1+}\_ : \mathbb{N} \to \mathit{NT}\ \_\ \mathit{Unit} \\ &g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\ &g\ (\mathrm{a}^{1+}\ \mathrm{zero}) \qquad\ = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ \mathit{return}\ \mathrm{unit} \\ &g\ (\mathrm{a}^{1+}\ (\mathrm{suc}\ n)) = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ !\,(\mathrm{a}^{1+}\ n) \end{aligned}$$

- Careful! Types can become really complicated:

$$\mathit{nt} : (n : \mathbb{N}) \to \mathit{NT}\ (f\ \ n)\ \mathit{Unit}$$

Expressive power?

- Can define grammars with an infinite number of non-terminals:

$$\begin{aligned} &\mathbf{data}\ \mathit{NT} : \mathit{ParserType}\ \mathbf{where} \\ &\quad \mathrm{a}^{1+}\_ : \mathbb{N} \to \mathit{NT}\ \_\ \mathit{Unit} \\ &g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\ &g\ (\mathrm{a}^{1+}\ \mathrm{zero}) \qquad\ = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ \mathit{return}\ \mathrm{unit} \\ &g\ (\mathrm{a}^{1+}\ (\mathrm{suc}\ n)) = \mathit{sym}\ \texttt{'a'}\ \circledast\!\!>\ !\,(\mathrm{a}^{1+}\ n) \end{aligned}$$

- The same warning applies when defining libraries.

▸ Libraries

Conclusions

- Structurally recursive descent parsing.
- Termination guaranteed.
- Errors caught at compile-time.
- Still feels like combinator parsing.
- More complicated types, but the overhead for the user is usually small.

Possible future work

- More efficient backend.
- Use backend which can handle left recursion ⇒ less complicated types.
    - But the types can be nice to have anyway.
    - And who needs left recursion?
      chainl is more high-level.

?

Extra slides

Defining a library: Non-terminals

The non-terminals are parameterised on the outer grammar's non-terminals:

```
data NT (nt : ParserType) : ParserType where
  many  : Parser tok nt (false, c) r → NT nt _ [r]
  many₁ : Parser tok nt (false, c) r → NT nt _ [r]
```

Defining a library: Combinators

Combinators parameterised on a *lib* constructor:

```
module Combinators (lib : NT nt i r → nt i r) where
  _⋆ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p⋆ = ! lib (many p)
  _⁺ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p⁺ = ! lib (many₁ p)
  library : NT nt i r → Parser tok nt i r
  library (many p)  = return [] ∣ p⁺
  library (many₁ p) = _∷_ $ p <⊛> p⋆
```

Defining a library: Combinators

Wrappers (to ease use of the library):

```
module Combinators (lib : NT nt i r → nt i r) where
  _⋆ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p⋆ = ! lib (many p)
  _⁺ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p⁺ = ! lib (many₁ p)
  library : NT nt i r → Parser tok nt i r
  library (many p)  = return [] ∣ p⁺
  library (many₁ p) = _∷_ $ p <⊛> p⋆
```

Defining a library: Combinators

Grammar (as before):

```
module Combinators (lib : NT nt i r → nt i r) where
  _⋆ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p ⋆  = ! lib (many p)
  _+ : Parser tok nt (false, c) r → Parser tok nt _ [r]
  p + = ! lib (many₁ p)
  library : NT nt i r → Parser tok nt i r
  library (many p)  = return [] ∣ p +
  library (many₁ p) = _∷_ $ p <*> p ⋆
```

Go back