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

$$
\begin{aligned}
\mathit{expr} \;=\;& \_+\_ \;\;\$\;\; \mathit{term} \;\;<\!*\;\; \mathit{sym}\;\text{'+'}\;\; <\!*\!>\;\; \mathit{expr} \\
\mid\;& \qquad\qquad\;\; \mathit{term} \\
\mathit{term} =\;& \ldots
\end{aligned}
$$

Simple example

$$
\begin{array}{lcl}
\mathit{expr} & = & \_{+}\_ \;\$\; \mathit{expr} \;<\!\!*\; \mathit{sym}\ \texttt{'+'} \;<\!\!*\!\!>\; \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & = & \ldots
\end{array}
$$

$\perp$

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:

$$\begin{aligned} expr = \ & \_+\_ \ \$ \ term \ \text{<*} \ sym \ \text{'+'} \ \text{<*>} \ expr \\ | \ & \qquad\qquad term \end{aligned}$$

- 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:

$$Grammar\ tok\ nt = nt\ res \rightarrow Parser\ tok\ nt\ res$$

- Rule out left recursion by restricting the types.

Examples

Example

- Non-terminals:

  data NT : $ParserType$ **where**
  expr : $NT \; \_ \; \mathbb{N}$
  term : $NT \; \_ \; \mathbb{N}$

- Result type: $\mathbb{N}$.
- 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

$$
\begin{array}{l}
g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\
g\ \text{expr}\ =\ \_{+}\_\ \ \$\ !\,\text{term}\ \ {<}{*}\ \ \mathit{sym}\ \text{'+'}\ \ {<}{*}{>}\ !\,\text{expr} \\
\qquad\quad\ \mid\qquad\qquad\ \ !\,\text{term} \\
g\ \text{term} = \ \ldots
\end{array}
$$

- 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

$$
\begin{array}{l}
\textbf{data } \mathit{NT} : \mathit{ParserType} \textbf{ where} \\
\quad \mathsf{lib} \quad : \mathit{L.Nonterminal}\ \mathit{NT}\ i\ r \rightarrow \mathit{NT}\ \_\ r \\
\quad \mathsf{expr} : \mathit{NT}\ \_\ \mathbb{N} \\
\quad \mathsf{term} : \mathit{NT}\ \_\ \mathbb{N} \\
\quad \mathsf{op} \quad : \mathit{NT}\ \_\ (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \\
\mathit{open}\ \mathit{L.Combinators}\ \mathsf{lib} \\
g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\
g\ (\mathsf{lib}\ p) = \mathit{libraryGrammar}\ p \\
g\ \mathsf{expr} \quad = \mathit{chainl}_1\ (!\,\mathsf{term})\ (!\,\mathsf{op}) \\
g\ \mathsf{term} \quad = \mathit{number} \mid \mathit{parenthesised}\ (!\,\mathsf{expr}) \\
g\ \mathsf{op} \quad\ \ = \_{+}\_ \ \ {<}\$\ \ \mathit{sym}\ \texttt{'+'} \\
\qquad\qquad \mid \ \_{-}\_ \ \ {<}\$\ \ \mathit{sym}\ \texttt{'-'}
\end{array}
$$

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

$$
\begin{aligned}
\mathit{parse} : \quad & \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \mathit{i}\ \mathit{result} \\
\rightarrow\ & \mathit{Grammar}\ \mathit{tok}\ \mathit{nt} \\
\rightarrow\ & [\mathit{tok}] \rightarrow [\mathit{result} \times [\mathit{tok}]]
\end{aligned}
$$

How does it work?

Indices

Parsers are indexed on two things:

$$\mathit{Index} = \mathit{Empty} \times \mathit{Corners}$$

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.

$$\begin{aligned}&\textbf{data}\ Corners : Set\ \textbf{where}\\&\quad \mathrm{leaf} : Corners\\&\quad \mathrm{step} : Corners \rightarrow Corners\\&\quad \mathrm{node} : Corners \rightarrow Corners \rightarrow Corners\end{aligned}$$

Some basic combinators

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

Some basic combinators

$$\_{<}{*}{>}\_ : \mathit{Parser}\ (e_1, c_1) \longrightarrow \mathit{Parser}\ (e_2, c_2) \longrightarrow$$
$$\mathit{Parser}\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \text{node}\ c_1\ c_2\ \textbf{else}\ c_1)$$

$$\_|\_ : \mathit{Parser}\ (e_1, c_1) \longrightarrow \mathit{Parser}\ (e_2, c_2) \longrightarrow$$
$$\mathit{Parser}\ (e_1 \vee e_2, \text{node}\ c_1\ c_2)$$

$$!\_ : \mathit{nt}\ (e, c) \longrightarrow \mathit{Parser}\ (e, \text{step}\ c)$$

Some basic combinators

$$\_{<}{*}{>}\_ : \mathit{Parser}\ (e_1, c_1) \to \mathit{Parser}\ (e_2, c_2) \to \mathit{Parser}\ (e_1 \wedge e_2,\ \textbf{if}\ e_1\ \textbf{then}\ \text{node}\ c_1\ c_2\ \textbf{else}\ c_1)$$

$$\_|\_ : \mathit{Parser}\ (e_1, c_1) \to \mathit{Parser}\ (e_2, c_2) \to \mathit{Parser}\ (e_1 \vee e_2, \text{node}\ c_1\ c_2)$$

$$!\_ : \mathit{nt}\ (e, c) \to \mathit{Parser}\ (e, \text{step}\ c)$$

This does not type check:

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

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

Some basic combinators

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

This works, though:

$$
\begin{aligned}
&\mathit{grammar} : \mathit{nt}\ (e, c)\ \mathit{res} \rightarrow \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (e, c)\ \mathit{res} \\
&\mathit{grammar}\ \text{rec} = \mathit{sym}\ c\ {*}{>}\ !\,\text{rec}
\end{aligned}
$$

Reason: *sym c* must consume a token.

Some basic combinators

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

Indirect left recursion also fails:

$$
\begin{aligned}
&\mathit{grammar} : \mathit{nt}\ (e, c)\ \mathit{res} \rightarrow \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (e, c)\ \mathit{res} \\
&\mathit{grammar}\ \text{rec} \quad\ = !\,\text{other}\ <\!\!*\!\!>\ \ldots \\
&\mathit{grammar}\ \text{other} = \mathit{many}\ p <\!\!*\!\!> !\,\text{rec}
\end{aligned}
$$

Indices can be useful anyway

- Parsing zero or more things:

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

- 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:

  data *NT* : *ParserType* **where**
  $\mathsf{a}^{1+}\_ : \mathbb{N} \rightarrow \mathit{NT}\ \_\ \mathit{Unit}$

  $g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT}$
  $g\ (\mathsf{a}^{1+}\ \mathsf{zero}) = \mathit{sym}\ \text{'a'}\ {*}{>}\ \mathit{return}\ \mathsf{unit}$
  $g\ (\mathsf{a}^{1+}\ (\mathsf{suc}\ n)) = \mathit{sym}\ \text{'a'}\ {*}{>}\ !\,(\mathsf{a}^{1+}\ n)$

- 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:

  $\textbf{data}\ \mathit{NT} : \mathit{ParserType}\ \textbf{where}$
  $\quad \mathsf{a}^{1+}\_ : \mathbb{N} \rightarrow \mathit{NT}\ \_\ \mathit{Unit}$

  $g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT}$
  $g\ (\mathsf{a}^{1+}\ \mathsf{zero}) = \mathit{sym}\ \texttt{'a'}\ \ {*}{>}\ \ \mathit{return}\ \mathsf{unit}$
  $g\ (\mathsf{a}^{1+}\ (\mathsf{suc}\ n)) = \mathit{sym}\ \texttt{'a'}\ \ {*}{>}\ \ !\,(\mathsf{a}^{1+}\ n)$

- Careful! Types can become really complicated:

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

Expressive power?

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{NT} : \mathit{ParserType}\ \textbf{where} \\
\quad \mathsf{a}^{1+}\_ : \mathbb{N} \rightarrow \mathit{NT}\ \_\ \mathit{Unit} \\
g : \mathit{Grammar}\ \mathit{Char}\ \mathit{NT} \\
g\ (\mathsf{a}^{1+}\ \mathsf{zero}) \qquad = \mathit{sym}\ \texttt{'a'}\ \ast\!>\ \mathit{return}\ \mathsf{unit} \\
g\ (\mathsf{a}^{1+}\ (\mathsf{suc}\ n)) = \mathit{sym}\ \texttt{'a'}\ \ast\!>\ !\,(\mathsf{a}^{1+}\ n)
\end{array}
$$

- 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 $\Rightarrow$ 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:

$$\textbf{data}\ \mathit{NT}\ (\mathit{nt} : \mathit{ParserType}) : \mathit{ParserType}\ \textbf{where}$$

$$\text{many}\ : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\text{false}, c)\ r \longrightarrow \mathit{NT}\ \mathit{nt}\ \_\ [r]$$

$$\text{many}_1 : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\text{false}, c)\ r \longrightarrow \mathit{NT}\ \mathit{nt}\ \_\ [r]$$

Defining a library: Combinators

Combinators parameterised on a *lib* constructor:

$$
\begin{aligned}
&\textbf{module}\ \mathit{Combinators}\ (\mathit{lib} : \mathit{NT}\ \mathit{nt}\ i\ r \to \mathit{nt}\ i\ r)\ \textbf{where}\\
&\quad \_^{\star}\ : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\mathrm{false}, c)\ r \to \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \_\ [r]\\
&\quad p^{\star}\ = !\ \mathit{lib}\ (\mathrm{many}\ p)\\
&\quad \_^{+}\ : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\mathrm{false}, c)\ r \to \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \_\ [r]\\
&\quad p^{+} = !\ \mathit{lib}\ (\mathrm{many}_1\ p)\\
&\quad \mathit{library} : \mathit{NT}\ \mathit{nt}\ i\ r \to \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ i\ r\\
&\quad \mathit{library}\ (\mathrm{many}\ p)\ \ = \mathit{return}\ [\,]\ \mid\ p^{+}\\
&\quad \mathit{library}\ (\mathrm{many}_1\ p) = \ \_{::}\_\ \ \$\ p\ <\!\ast\!>\ p^{\star}
\end{aligned}
$$

Defining a library: Combinators

Wrappers (to ease use of the library):

$$
\begin{array}{l}
\textbf{module } \mathit{Combinators}\ (\mathit{lib} : \mathit{NT}\ \mathit{nt}\ i\ r \rightarrow \mathit{nt}\ i\ r)\ \textbf{where} \\
\quad \_^{\star}\ : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\mathsf{false}, c)\ r \rightarrow \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \_\ [r] \\
\quad p^{\star}\ = !\, \mathit{lib}\ (\mathsf{many}\ p) \\
\quad \_^{+} : \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ (\mathsf{false}, c)\ r \rightarrow \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ \_\ [r] \\
\quad p^{+} = !\, \mathit{lib}\ (\mathsf{many}_1\ p) \\
\quad \mathit{library} : \mathit{NT}\ \mathit{nt}\ i\ r \rightarrow \mathit{Parser}\ \mathit{tok}\ \mathit{nt}\ i\ r \\
\quad \mathit{library}\ (\mathsf{many}\ p)\ \ = \mathit{return}\ [\,]\ \mid\ p^{+} \\
\quad \mathit{library}\ (\mathsf{many}_1\ p) = \ \_{::}\_\ \ \$\ \ p\ <\!\!*\!\!>\ p^{\star}
\end{array}
$$

Defining a library: Combinators

Grammar (as before):

module *Combinators* $(lib : NT\ nt\ i\ r \rightarrow nt\ i\ r)$ **where**

$$
\begin{aligned}
&\_^{\star}\ : Parser\ tok\ nt\ (\mathsf{false}, c)\ r \rightarrow Parser\ tok\ nt\ \_\ [r] \\
&p^{\star}\ = !\, lib\ (\mathsf{many}\ p) \\
&\_^{+}\ : Parser\ tok\ nt\ (\mathsf{false}, c)\ r \rightarrow Parser\ tok\ nt\ \_\ [r] \\
&p^{+}\ = !\, lib\ (\mathsf{many}_1\ p) \\
&library : NT\ nt\ i\ r \rightarrow Parser\ tok\ nt\ i\ r \\
&library\ (\mathsf{many}\ p) = return\ [\,]\ \mid\ p^{+} \\
&library\ (\mathsf{many}_1\ p) = \_{::}\_\ \$\ p\ <\!\ast\!>\ p^{\star}
\end{aligned}
$$

Go back