GLL Parsing with Flexible Combinators

L. Thomas van Binsbergen,
Elizabeth Scott, and Adrian Johnstone

Royal Holloway, University of London
ltvanbinsbergen@acm.org

5 November, 2018

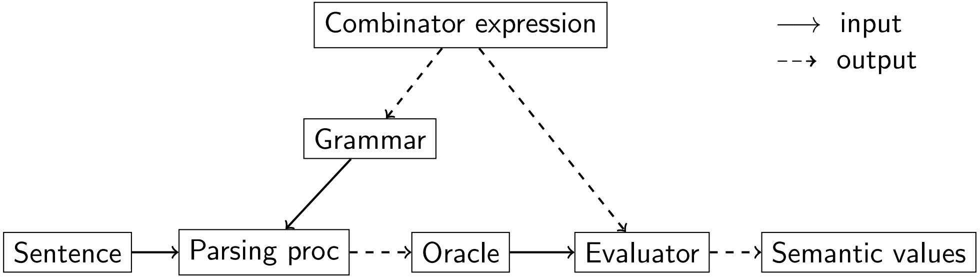
http://hackage.haskell.org/package/gll

Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle

Tom Ridge

University of Leicester

Library Architecture (Ridge 2014)



- Parser is replaceable
- Similar suggestion by [Ljunglöf, 2002]

Contributions

1. Functional description and implementation of GLL parsing:
   - All datastructures are basic sets/relations
   - Recursive descent extended to GLL
2. Grammar combinators without grammar binarisation:
   - Combinator expressions evaluate to a grammar object
   - This grammar is an argument to parsing procedure
3. Emperical evaluation on real-world grammars:
   - Demonstrates "acceptable" runtimes on ANSI-C, Caml Light, CBS
   - Significant speed-ups achieved by avoiding binarisation

GLL parsing

1) Recursive descent parsing

- Every nonterminal is implemented by a *parse function*
- Every parse function has a *branch* for every alternate of the nonterminal
- Every branch is a sequence of:
    - calls to parse functions
    - code matching terminals

GLL parsing

1) Recursive descent parsing

- Every nonterminal is implemented by a *parse function*
- Every parse function has a *branch* for every alternate of the nonterminal
- Every branch is a sequence of:
  - calls to parse functions
  - code matching terminals

2) We consider parse functions that:

- have a parameter holding an index k into the input string (pivot)
- have a local variable remembering the initial pivot value l (left extent)
- return the value r (right extent) held by the parameter at the end of a branch

GLL parsing

2) We consider parse functions that:

- have a parameter holding an index k into the input string (pivot)
- have a local variable remembering the initial pivot value l (left extent)
- return the value r (right extent) held by the parameter at the end of a branch

3) Abstract representation

A *descriptor* $(X ::= \alpha \cdot \beta, l, k)$ models the state of a parse

A *commencement* (X, l) models a (parse) function call

A *continuation* $(X ::= \alpha Y \cdot \beta, l)$ models a return context

GLL parsing

3) Abstract representation

A *descriptor* $(X ::= \alpha \cdot \beta, l, k)$ models the state of a parse

A *commencement* (X, l) models a (parse) function call

A *continuation* $(X ::= \alpha Y \cdot \beta, l)$ models a return context

4) Descriptor processing

Process encountered descriptors in any order, exactly once, starting with $(S ::= \cdot\alpha, 0, 0)$

There are three forms of descriptors:

- $(X ::= \alpha \cdot t\beta, l, k)$ with t terminal — **match** action
- $(X ::= \alpha \cdot Y\beta, l, k)$ with Y nonterminal — **descend**/**skip** action
- $(Y ::= \delta\cdot, l, r)$ — **ascend** action

GLL parsing

4) Descriptor processing

Process encountered descriptors in any order, exactly once, starting with $(S ::= \cdot\alpha, 0, 0)$

There are three forms of descriptors:

- $(X ::= \alpha \cdot t\beta, l, k)$ with t terminal — **match** action
- $(X ::= \alpha \cdot Y\beta, l, k)$ with Y nonterminal — **descend**/**skip** action
- $(Y ::= \delta\cdot, l, r)$ — **ascend** action

5) GLL datatypes

The set $\mathcal{U}$ contains all descriptors processed so far

The relation $\mathcal{P}$ pairs commencements with right extents

The relation $\mathcal{G}$ pairs commencements with continuations

GLL parsing

5) GLL datatypes

The set $\mathcal{U}$ contains all descriptors processed so far

The relation $\mathcal{P}$ pairs commencements with right extents

The relation $\mathcal{G}$ pairs commencements with continuations

match

$$(X ::= \alpha \cdot t\beta, l, k) \quad \rightarrow \quad (X ::= \alpha t \cdot \beta, l, k+1)$$

pre-conditions:

- t is the k'th terminal in the input string

post-conditions:

- $(X ::= \alpha \cdot t\beta, l, k) \in \mathcal{U}$

GLL parsing

descend

$$
\begin{aligned}
(X ::= \alpha \cdot Y\beta, l, k) \quad &\rightarrow \quad (Y ::= \cdot\delta_1, k, k) \\
&\dots \\
&\rightarrow \quad (Y ::= \cdot\delta_i, k, k)
\end{aligned}
$$

pre-conditions:

- $Y ::= \delta_i$ is in the grammar, for all i
- There is no r such that $((Y, k), r) \in \mathcal{P}$

post-conditions:

- Possible new continuation: $((Y, k), (X ::= \alpha Y \cdot \beta, l)) \in \mathcal{G}$
- $(X ::= \alpha \cdot Y\beta, l, k) \in \mathcal{U}$

GLL parsing

skip

$$\begin{aligned}(X ::= \alpha \cdot Y\beta, l, k) &\rightarrow (X ::= \alpha Y \cdot \beta, l, r_1) \\ &\dots \\ &\rightarrow (X ::= \alpha Y \cdot \beta, l, r_j)\end{aligned}$$

pre-conditions:

- For all $1 \leqslant i \leqslant j$, we have $((Y, k), r_i) \in \mathcal{P}$ (at least one)

post-conditions:

- Possible new continuation: $((Y, k), (X ::= \alpha Y \cdot \beta, l)) \in \mathcal{G}$
- $(X ::= \alpha \cdot Y\beta, l, k) \in \mathcal{U}$

ascend

$$(Y ::= \delta\cdot, l, r) \quad \to \quad (X ::= \alpha_1 Y \cdot \beta_1, l_1, r)$$
$$\dots$$
$$\to \quad (X ::= \alpha_i Y \cdot \beta_i, l_j, r)$$

pre-conditions:

- For all $1 \leqslant i \leqslant j$, we have $((Y, l), (X ::= \alpha_i Y \cdot \beta_i, l_i)) \in \mathcal{G}$

post-conditions:

- Possible new right extent: $((Y, l), r) \in \mathcal{P}$
- $(Y ::= \delta\cdot, l, r) \in \mathcal{U}$

Oracle construction

$$(X ::= \alpha \cdot s\beta, l, k) \in \mathcal{U} \quad \& \quad (X ::= \alpha s \cdot \beta, l, r) \in \mathcal{U}$$

gives

$$(X ::= \alpha s \cdot \beta, l, k, r) \in \mathcal{O}$$

$$(Y ::= \delta \cdot, l, r) \quad \textbf{with } l = r, \delta = \epsilon$$

gives

$$(Y ::= \delta \cdot, l, l, l) \in \mathcal{O}$$

Contributions

1. Functional description and implementation of GLL parsing:
   - All datastructures are basic sets/relations
   - Recursive descent extended to GLL
2. Grammar combinators without grammar binarisation:
   - Combinator expressions evaluate to a grammar object
   - This grammar is an argument to parsing procedure
3. Emperical evaluation on real-world grammars:
   - Demonstrates "acceptable" runtimes on ANSI-C, Caml Light, CBS
   - Significant speed-ups achieved by avoiding binarisation

Parser combinators

$$\begin{array}{llll} term & :: Eq\ t & \Rightarrow t & \rightarrow Parser \\ epsilon & :: Parser & & \\ (\langle * \rangle) & :: Parser & \rightarrow Parser & \rightarrow Parser \\ (\langle | \rangle) & :: Parser & \rightarrow Parser & \rightarrow Parser \end{array}$$

Example $T ::= (\ A\ )\quad A ::= \epsilon \mid M\ a\quad M ::= \epsilon \mid M\ a\ ,$

$$\begin{array}{ll} pT & = term\ \texttt{'('}\ \langle * \rangle\ pA\ \langle * \rangle\ term\ \texttt{')'} \\ pA & = epsilon\ \langle | \rangle\ pM\ \langle * \rangle\ term\ \texttt{'a'} \\ pM & = epsilon\ \langle | \rangle\ pM\ \langle * \rangle\ term\ \texttt{'a'}\ \langle * \rangle\ term\ \texttt{','} \end{array}$$

Grammar combinators

$$
\begin{aligned}
&\mathit{term} && :: \mathit{Eq}\ t \Rightarrow t \rightarrow \mathit{Grammar} \\
&\mathit{epsilon} && :: \mathit{Grammar} \\
&(\langle * \rangle) && :: \mathit{Grammar} \rightarrow \mathit{Grammar} \rightarrow \mathit{Grammar} \\
&(\langle | \rangle) && :: \mathit{Grammar} \rightarrow \mathit{Grammar} \rightarrow \mathit{Grammar}
\end{aligned}
$$

Grammar extraction

- Expressions yield at most two productions with at most two symbols in rhs

$$nt(x) = \texttt{"("} +\!\!+\ nt(l) +\!\!+ \texttt{"*"} +\!\!+\ nt(r) +\!\!+ \texttt{")"} \qquad \textbf{if } x = l \langle * \rangle r$$

$$nt(y) = \texttt{"("} +\!\!+\ nt(p) +\!\!+ \texttt{"|"} +\!\!+\ nt(q) +\!\!+ \texttt{")"} \qquad \textbf{if } y = p \langle | \rangle q$$

productions: $nt(x) ::= nt(l)nt(r)$, $nt(y) ::= nt(p)$, $nt(y) ::= nt(q)$

Grammar combinators

$$
\begin{aligned}
&\mathit{nterm} &&:: \mathit{String} && && \rightarrow \mathit{Grammar}\\
&\mathit{term} &&:: \mathit{Eq}\ t && \Rightarrow t && \rightarrow \mathit{Grammar}\\
&\mathit{epsilon} &&:: \mathit{Grammar} && && \\
&(\langle * \rangle) &&:: \mathit{Grammar} && \rightarrow \mathit{Grammar} && \rightarrow \mathit{Grammar}\\
&(\langle | \rangle) &&:: \mathit{Grammar} && \rightarrow \mathit{Grammar} && \rightarrow \mathit{Grammar}
\end{aligned}
$$

Grammar extraction

- Expressions yield at most two productions with at most two symbols in rhs

$$nt(x) = \texttt{"("} +\!\!+\ nt(l) +\!\!+ \texttt{"*"} +\!\!+\ nt(r) +\!\!+ \texttt{")"} \qquad \textbf{if } x = l\ \langle * \rangle\ r$$

$$nt(y) = \texttt{"("} +\!\!+\ nt(p) +\!\!+ \texttt{"|"} +\!\!+\ nt(q) +\!\!+ \texttt{")"} \qquad \textbf{if } y = p\ \langle | \rangle\ q$$

productions: $nt(x) ::= nt(l)nt(r)$, $nt(y) ::= nt(p)$, $nt(y) ::= nt(q)$

BNF combinators

$$
\begin{array}{lll}
(\langle ::= \rangle) & :: \mathit{String} \rightarrow \mathit{Choice}_{\mathrm{EX}} & \rightarrow \mathit{Symb}_{\mathrm{EX}} \\
\mathit{term} & :: \mathit{Eq}\ t \quad \Rightarrow t & \rightarrow \mathit{Symb}_{\mathrm{EX}} \\
(\langle ** \rangle) & :: \mathit{Seq}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Seq}_{\mathrm{EX}} & \\
\mathit{seqStart} & :: \mathit{Seq}_{\mathrm{EX}} & \\
(\langle || \rangle) & :: \mathit{Choice}_{\mathrm{EX}} \rightarrow \mathit{Seq}_{\mathrm{EX}} \rightarrow \mathit{Choice}_{\mathrm{EX}} & \\
\mathit{altStart} & :: \mathit{Choice}_{\mathrm{EX}} &
\end{array}
$$

Example $T ::= (\ A\ )$

$$
\begin{array}{l}
gT = \texttt{"T"}\ \langle ::= \rangle\ \mathit{altStart}\ \langle || \rangle\ \mathit{seqStart}\ \langle ** \rangle\ \mathit{term}\ \texttt{'('}\ \langle ** \rangle\ gA\ \langle ** \rangle\ \mathit{term}\ \texttt{')'} \\
gA = \ldots
\end{array}
$$

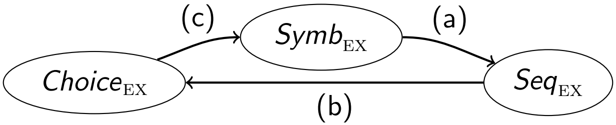
Flexible BNF combinators

$$
\begin{array}{l}
(\mathit{IsSeq}\ \mathit{seq}, \mathit{IsCh}\ \mathit{ch}, \mathit{IsSymb}\ \mathit{symb}) \Rightarrow \\
\quad (\langle{::=}\rangle) \ :: \mathit{String} \rightarrow \mathit{ch} \rightarrow \mathit{Symb}_{\mathrm{EX}} \\
\quad \mathit{term} \ :: \mathit{Eq}\ t \Rightarrow t \rightarrow \mathit{Symb}_{\mathrm{EX}} \\
\quad (\langle{**}\rangle) \ :: \mathit{seq} \rightarrow \mathit{symb} \rightarrow \mathit{Seq}_{\mathrm{EX}} \\
\quad \mathit{seqStart} :: \mathit{Seq}_{\mathrm{EX}} \\
\quad (\langle||\rangle) \ :: \mathit{ch} \rightarrow \mathit{seq} \rightarrow \mathit{Choice}_{\mathrm{EX}} \\
\quad \mathit{altStart} :: \mathit{Choice}_{\mathrm{EX}}
\end{array}
$$

Example $T ::= (A)$

$$
\begin{array}{l}
gT = \texttt{"T"}\ \langle{::=}\rangle\ \mathit{term}\ \texttt{'('}\ \langle{**}\rangle\ gA\ \langle{**}\rangle\ \mathit{term}\ \texttt{')'} \\
gA = \ldots
\end{array}
$$

Conversions

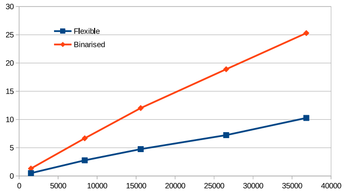


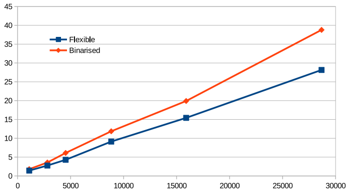
instance *IsSeq* Seq_{EX}	**where** ...	-- id
instance *IsSeq* $Symb_{\mathrm{EX}}$	**where** ...	-- (a)
instance *IsSeq* $Choice_{\mathrm{EX}}$	**where** ...	-- (a) $\circ$ (c)
instance *IsCh* $Choice_{\mathrm{EX}}$	**where** ...	-- id
instance *IsCh* Seq_{EX}	**where** ...	-- (b)
instance *IsCh* $Symb_{\mathrm{EX}}$	**where** ...	-- (b) $\circ$ (a)
instance *IsSymb* $Symb_{\mathrm{EX}}$	**where** ...	-- id
instance *IsSymb* $Choice_{\mathrm{EX}}$	**where** ...	-- (c)
instance *IsSymb* Seq_{EX}	**where** ...	-- (c) $\circ$ (b)

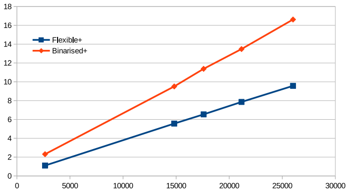
Contributions

1. Functional description and implementation of GLL parsing:
   - All datastructures are basic sets/relations
   - Recursive descent extended to GLL
2. Grammar combinators without grammar binarisation:
   - Combinator expressions evaluate to a grammar object
   - This grammar is an argument to parsing procedure
3. Emperical evaluation on real-world grammars:
   - Demonstrates "acceptable" runtimes on ANSI-C, Caml Light, CBS
   - Significant speed-ups achieved by avoiding binarisation

Evaluation







Claims

- Running times show that the approach is practical

  Although the emphasis has been on correctness

- Avoiding binarisation improves running times

  Syntax descriptions have not been manipulated to benefit evaluation

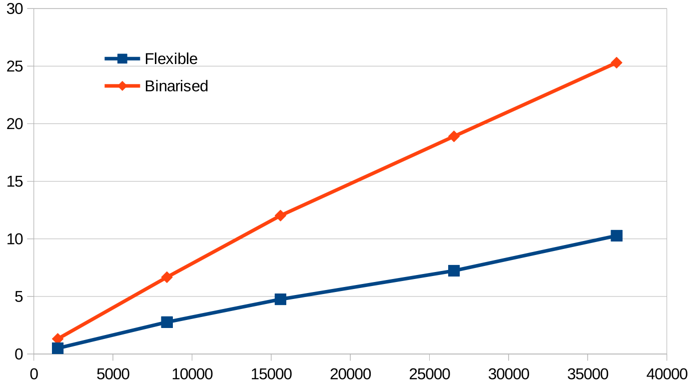
Binarising BNF combinators

$$(\ \langle ::= \rangle_{\mathrm{BIN}}\ ) :: \mathit{String} \rightarrow \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}}$$
$$(\ \langle ::= \rangle_{\mathrm{BIN}}\ ) = (\ \langle ::= \rangle\ )$$

$$(\ \langle || \rangle_{\mathrm{BIN}}\ ) :: \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}}$$
$$p\ \langle || \rangle_{\mathrm{BIN}}\ q = \mathit{toSymb}\ (p\ \langle || \rangle\ q)$$

$$(\ \langle ** \rangle_{\mathrm{BIN}}\ ) :: \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}} \rightarrow \mathit{Symb}_{\mathrm{EX}}$$
$$p\ \langle ** \rangle_{\mathrm{BIN}}\ q = \mathit{toSymb}\ (p\ \langle ** \rangle\ q)$$

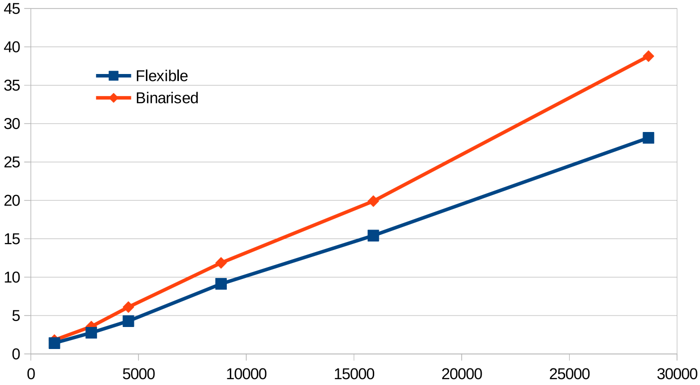
Parsing: ANSI-C



Binarised: 690 nonterminals, 848 alternates
Flexible: 71 nonterminals, 229 alternates

2.4-2.6x speed-up (with lookahead)

Parsing: Caml Light

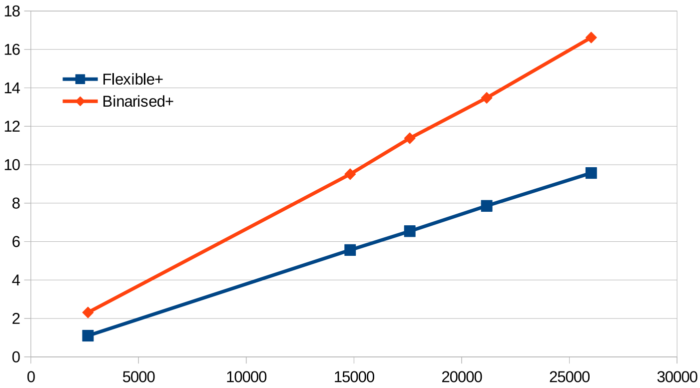


Binarised: 580 nonterminals, 731 alternates

Flexible: 134 nonterminals, 285 alternates

1.3-1.4x speed-up (with lookahead)

Parsing and printing: Component-Based Semantics



Binarised: 640 nonterminals, 771 alternates

Flexible: 126 nonterminals, 257 alternates

1.7-2.1x speed-up (with lookahead)

Practical reflection

- An EDSL for describing context-free grammars based on ‘BNF combinators’
- Parsers with on-the-fly semantics available for described grammars
- Generalised parsing certainly simplifies SLE
- Library suitable for our purpose: reference interpreters for programming languages
- Caveats:
  - Disambiguation mostly ad-hoc
  - Manual nonterminal insertion problematic in some cases

GLL Parsing with Flexible Combinators

L. Thomas van Binsbergen,
Elizabeth Scott, and Adrian Johnstone

Royal Holloway, University of London
ltvanbinsbergen@acm.org

5 November, 2018

http://hackage.haskell.org/package/gll

http://ltvanbinsbergen.nl/thesis

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 ...	