Combinators for Generalised Parsing

L. Thomas van Binsbergen

Royal Holloway, University of London

October 6, 2016
1. Conventional Parsing
   - Grammars and Parsing
   - Recursive Descent Parsing

2. Conventional Parser Combinators
   - Monadic combinators in Haskell

3. Generalised Parsing
   - Problems with RD Parsing
   - GLL algorithm

4. Combinators for Generalised Parsing
   - Explicit grammar information
   - GLL Combinators
Section 1

Conventional Parsing
There are two types of symbols:
- Atomic terminals
- Nonterminals, expanding according to productions

A grammar is a set of productions

Grammar \( x ::= \alpha \quad x ::= \beta \quad y ::= \gamma \quad \ldots \)

specifies that nonterminal \( x \) expands to symbol sequence \( \alpha \) or \( \beta \)

Does \( x \) derive \( I \)? Can we keep expanding \( x \) till equal to \( I \)?
• Terminals: + ∗ ( ) INT

• Productions:

\[
e_0 ::= e_1 \\ e_1 ::= e_2 \\ e_2 ::= \text{INT} \\
\]

\[
e_0 ::= e_1 + e_0 \\ e_1 ::= e_2 * e_1 \\ e_2 ::= (e_0) \\
\]

• Is there a full expansion of \(e_0\) equal to \(1+2*3\)?
• Terminals: $+ \ast ( ) INT$

• Productions:

\[
\begin{align*}
e_0 &::= e_1 & e_1 &::= e_2 & e_2 &::= INT \\
e_0 &::= e_1 + e_0 & e_1 &::= e_2 \ast e_1 & e_2 &::= (e_0)
\end{align*}
\]

• Is there a full expansion of $e_0$ equal to $1+2\ast3$?

• A yes/no answer is only recognition!

• A parser provides proof in the form of a parse tree

• Or even better: a parser performs evaluation on the fly
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

- Let’s compose parse functions with a choice and sequence op!
- Grammar:

\[
\begin{align*}
e_0 & ::= e_1 \\
e_0 & ::= e_1 + e_0 \\
e_1 & ::= e_2 \\
e_1 & ::= e_2 * e_1 \\
e_2 & ::= INT \\
e_2 & ::= (e_0)
\end{align*}
\]

- Input string: 1+2*3
- Current index: 0
- Current stack: []

  \[
  \cdot e_0
  \]
• Grammar:

\[
\begin{align*}
e_0 & ::= e_1 \\
e_0 & ::= e_1 + e_0 \\
e_0 & ::= ( e_0 )
\end{align*}
\]

\[
\begin{align*}
e_1 & ::= e_2 \\
e_1 & ::= e_2 * e_1 \\
e_2 & ::= \text{INT}
\end{align*}
\]

• Input string: 1+2*3
• Current index: 0
• Current stack: [ · e_0]

\[
e_0 ::= ·e_1 + e_0
\]
Grammar:

\[
e_0 ::= e_1 \quad e_1 ::= e_2 \quad e_2 ::= INT
\]

\[
e_0 ::= e_1 + e_0 \quad e_1 ::= e_2\ast e_1 \quad e_2 ::= (e_0)
\]

- Input string: 1+2*3
- Current index: 0
- Current stack: \([e_0 ::= \cdot e_1 + e_0, \cdot e_0]\)

\[
e_1 ::= \cdot e_2
\]
Grammar:

\[ e_0 ::= e_1 \]
\[ e_1 ::= e_2 \]
\[ e_2 ::= \text{INT} \]
\[ e_0 ::= e_1 + e_0 \]
\[ e_1 ::= e_2 * e_1 \]
\[ e_2 ::= (e_0) \]

Input string: 1+2*3
Current index: 0
Current stack: \([e_1 ::= \cdot e_2, e_0 ::= \cdot e_1 + e_0, \cdot e_0]\]

\[ e_2 ::= \cdot \text{INT} \]
• Grammar:

\[ e_0 ::= e_1 \quad e_1 ::= e_2 \quad e_2 ::= INT \]
\[ e_0 ::= e_1 + e_0 \quad e_1 ::= e_2 * e_1 \quad e_2 ::= (e_0) \]

• Input string: 1+2*3
• Current index: 1
• Current stack: \([e_1 ::= \cdot, e_0 ::= \cdot e_1 + e_0, \cdot e_0]\)
- Grammar:

  \[
  \begin{align*}
  e_0 &::= e_1 \\
  e_0 &::= e_1 + e_0 \\
  e_1 &::= e_2 \\
  e_1 &::= e_2 \cdot e_1 \\
  e_2 &::= INT \\
  e_2 &::= ( e_0 )
  \end{align*}
  \]

- Input string: 1+2*3
- Current index: 1
- Current stack: \([e_0 ::= \cdot e_1 + e_0, \cdot e_0]\)
• Grammar:

\[
\begin{align*}
    e_0 & ::= e_1 \\
    e_0 & ::= e_1 + e_0 \\
    e_1 & ::= e_2 \\
    e_1 & ::= e_2 * e_1 \\
    e_2 & ::= \text{INT} \\
    e_2 & ::= (e_0)
\end{align*}
\]

• Input string: 1+2*3

• Current index: 1

• Current stack: [· e_0]

\[
e_0 ::= e_1 + e_0
\]
• Grammar:

\[
\begin{align*}
  e_0 & ::= e_1 \\
  e_0 & ::= e_1 + e_0 \\
  e_1 & ::= e_2 \\
  e_1 & ::= e_2 \ast e_1 \\
  e_2 & ::= \text{INT} \\
  e_2 & ::= (e_0)
\end{align*}
\]

• Input string: 1+2\ast3

• Current index: 2

• Current stack: \([ \cdot e_0]\)

\[
e_0 ::= e_1 + \cdot e_0
\]
• Grammar:

\[
\begin{align*}
e_0 &::= e_1 \\
        &::= e_1 + e_0 \\
        &::= e_2 \cdot e_0 \\
\end{align*}
\]

\[
\begin{align*}
e_1 &::= e_2 \\
        &::= e_2 \cdot e_1 \\
\end{align*}
\]

\[
e_2 ::= \text{INT} \\
\]

• Input string: 1\(\text{+}\)2\(\text{*}\)3

• Current index: 5

• Current stack: [ · e_0 ]

\[
e_0 ::= e_1 + e_0 .
\]
• Grammar:

\[
\begin{align*}
    e_0 & ::= e_1 \\
    e_0 & ::= e_1 + e_0 \\
    e_1 & ::= e_2 \\
    e_1 & ::= e_2 * e_1 \\
    e_2 & ::= \text{INT} \\
    e_2 & ::= (e_0)
\end{align*}
\]

• Input string: 1+2*3
• Current index: 5
• Current stack: []
Section 2

Conventional Parser Combinators
Section 3

Generalised Parsing
Problems with Recursive Descent Parsing

- Nontermination: if descending 
  \((e_0, 0)\) requires descending 
  \((e_0, 0)\)

- Explosion of choices:
  An algorithm that gives up after a wrong choice is incomplete.
  An algorithm that backtracks suffers:
  Exponentially many alternatives may have to be explored.

Diagram:
\[
\begin{array}{c}
(e_0, 0) \\
\begin{array}{c}
\text{a}_1 \mid \text{a}_2 \mid \text{a}_3
\end{array}
\end{array}
\]
Problems with Recursive Descent Parsing

- Nontermination: if descending $(e_0, 0)$ requires descending $(e_0, 0)$
- Explosion of choices:
  An algorithm that gives up after a wrong choice is incomplete
  An algorithm that backtracks suffers: Exponentially many alternatives may have to be explored
Choosing between alternatives

1. Pick an alternative and stick with it
Choosing between alternatives

1. Pick an alternative and stick with it
2. Backtrack to the first alternative that succeeds
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Problems with RD Parsing

GLL algorithm

Choosing between alternatives

1. Pick an alternative and stick with it
2. Backtrack to the first alternative that succeeds
3. Pick all alternatives (and accumulate the results)
Problems with Recursive Descent

- Nontermination: if descending \((e_0, 0)\) requires descending \((e_0, 0)\)
Problems with Recursive Descent

- Nontermination: if descending \((e_0, 0)\) requires descending \((e_0, 0)\)
- Explosion of choices:
  - An algorithm that gives up after a wrong choice is incomplete
  - An algorithm that backtracks suffers: 
    *Exponentially many alternatives may have to be explored*
- Cocke-Younger-Kasami (CYK)
- Earley parsing (1970)
- GLR (Tomita 1984 - Scott & Johnstone 2007)

**GLL algorithm - Generalised RD parsing**
- Explore all alternatives
- No parse function is executed twice with the same arguments
- Can be implemented in $O(n^3)$
- Computes an efficient representation of all parse trees
Johnson’s memoisation

- Combinators are defined in *continuation passing style*
- Memoise parse functions by storing for each pivot argument:
  - Result indices
  - Return positions (continuations)
Problems with RD Parsing

GLL algorithm

\[(e_0, 0)\]

\[a_1 \mid a_2 \mid a_3\]
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Problems with RD Parsing
GLL algorithm

$$c_1 \rightarrow (e_0, 0)$$

All arriving continuations are applied to all result indices

No duplication:
Only the first application of $(e_0, 0)$ applies alternatives

No continuation is applied to the same result twice
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Problems with RD Parsing
GLL algorithm

\[
\begin{align*}
&c_1 \\
&\rightarrow (e_0, 0) \\
&a_1 | a_2 | a_3 \\
&\leftarrow r_1
\end{align*}
\]
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Problems with RD Parsing
GLL algorithm

\[(c_2 r_1)\]

\[
\begin{array}{c}
(c_1) \\
\downarrow \\
\hline \\
\begin{array}{c}
\text{c_1} \\
\text{c_2}
\end{array}
\end{array}
\]

\[a_1 \mid a_2 \mid a_3\]

\[
\begin{array}{c}
\text{c_1} \\
\downarrow \\
\text{(e_0, 0)} \\
\downarrow \\
\text{c_2}
\end{array}
\]

\[
\begin{array}{c}
\downarrow \\
r_1
\end{array}
\]

\[r_1\]

\[r_1\]

No duplication:
Only the first application of \((e_0, 0)\) applies alternatives
No continuation is applied to the same result twice
All arriving continuations are applied to all result indices

No duplication:

Only the first application of \((e_0, 0)\) applies alternatives

No continuation is applied to the same result twice
All arriving continuations are applied to all result indices

No duplication:

- Only the first application of \((e_0, 0)\) applies alternatives
- No continuation is applied to the same result twice
Whenever a symbol has been matched (say $e_1$), remember:

- Which production we are currently matching: $e_0 ::= e_1 \cdot + e_0$
- The pivot when we started matching this production
- The pivot when we started matching the symbol
- The pivot after matching the symbol
Section 4

Combinators for Generalised Parsing
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide *any* grammar info
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide *any* grammar info
• We require unique identification of *recursive* parse functions
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide any grammar info
• We require unique identification of recursive parse functions
• We require unique identification of ‘costly’ parse functions
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide any grammar info
• We require unique identification of recursive parse functions
• We require unique identification of ‘costly’ parse functions
• All we need is...
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide any grammar info
• We require unique identification of recursive parse functions
• We require unique identification of ‘costly’ parse functions
• All we need is... observable sharing
• GLL requires unique identifiers for symbols
• Conventional P.C. do not provide any grammar info
• We require unique identification of recursive parse functions
• We require unique identification of ‘costly’ parse functions
• All we need is... observable sharing

Observable sharing
  • Simple pure ‘solution’: Ask the programmer!
  • The way we obtain observable sharing influences how derived combinator are defined
Existing Libraries

- XSaiga (Haskell) - Frost et al. (2008)
- P3 (OCaml) - Ridge (2014)
- Meerkat (Scala) - Izmaylova et al. (2016)
- GLL Combinators (Haskell)

- Grammar Combinators (Haskell) - Devriese et al. (2011)
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Explicit grammar information
GLL Combinators

GLL Library Overview

Derivation Info

GLL Parser
Grammar
Expression

String
Evaluator
Semantic Values

disambiguation??
Issues for library designers

- How to provide disambiguation options?
- Grammar transforming? Top-down? Bottom-up?
- Defining additional elementary combinators is hard but desired

A user’s perspective

- Harder to define derived combinators (observable sharing)
- Even harder to define elementary combinators
- Monadic parsing is not an option (post-parse disambiguation)

\[
\text{multiple} :: \text{BNF } t \ a \rightarrow \text{BNF } t \ [ a ] \\
\text{multiple } x = \\
\quad \text{let } \text{fresh} = \text{mkNt } x \ "*" \\
\quad \text{in } \text{fresh } \langle ::= \rangle ( : ) \langle $\$ \rangle x \langle ** \rangle \text{multiple } x \ -- \ x ::= x \ x^* \\
\quad \langle || \rangle \text{ satisfy } [ ] \\
\quad \langle || \rangle \text{ satisfy } [ ] \ -- \ x ::= \ #
\]
Conventional Parsing
Conventional Parser Combinators
Generalised Parsing
Combinators for Generalised Parsing

Explicit grammar information
GLL Combinators

Combinators for Generalised Parsing

L. Thomas van Binsbergen

Royal Holloway, University of London

October 6, 2016
• With the gll-package you build grammar expressions

```
type SymbExpr t a = ...
type AltExpr t a = ...
type AltsExpr t a = ...
```

```
 Antar (⟨::=⟩) :: String → AltsExpr t a → SymbExpr t a
 Antar term :: t → SymbExpr t t
 Antar (⟨$$⟩) :: (a → b) → SymbExpr t a → AltExpr t b
 Antar (⟨**⟩) :: AltExpr t (a → b) → SymbExpr t a → AltExpr t b
 Antar empty :: AltsExpr t a
 Antar (⟨||⟩) :: AltExpr t a → AltsExpr t a → AltsExpr t a
```
With the gll-package you build *grammar expressions*

```haskell

```

| (IsAltExpr alt, HasAlts alts, IsSymbExpr s) ⇒
| (::<=>) :: String → alts t a → SymbExpr t a
| term :: t → SymbExpr t t
| ⟨$$⟩ :: (a → b) → s t a → AltExpr t b
| ⟨∗∗⟩ :: alt t (a → b) → s t a → AltExpr t b
| empty :: AltsExpr t a
| ⟨||⟩ :: alt t a → alts t a → AltsExpr t a
```
instance IsSymbExpr AltExpr where ...
instance IsSymbExpr SymbExpr where ...
instance IsSymbExpr AltsExpr where ...
instance HasAlts AltExpr where ...
instance HasAlts SymbExpr where ...
instance HasAlts AltsExpr where ...
instance IsAltExpr AltExpr where ...
instance IsAltExpr SymbExpr where ...
instance IsAltExpr AltsExpr where ...
Matching

subject to:

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

\[(x ::= \alpha \cdot t\beta, l, k) \rightarrow (x ::= \alpha t \cdot \beta, l, k + 1)\]

State Update

- New extended packed node \((x ::= \alpha, l, k, k + 1)\)
**Descending**

subject to:

- No previous result for \((y, k)\)
- \(y ::= \gamma_i\) is a valid production  
  \[\text{(for all } i\text{)}\]

\[
(x ::= \alpha \cdot y \beta, l, k) \rightarrow (y ::= \gamma_1, k, k)
\]

\[
\ldots
\]

\[
\rightarrow (y ::= \gamma_i, k, k)
\]

**State Update**

- Store \((x ::= \alpha y \cdot \beta, l)\) as a continuation for \((y, k)\)
Ascending

*subject to:*

- \((x ::= \alpha_i y \cdot \beta_i, l_i)\) is a continuation for \((y, k)\) (for all \(i\))

\[
\begin{align*}
(y ::= \gamma \cdot, k, r) & \rightarrow (x ::= \alpha_1 y \cdot \beta_1, l_1, r) \\
& \cdots \\
& \rightarrow (x ::= \alpha_i y \cdot \beta_i, l_i, r)
\end{align*}
\]

State Update

- Store \(r\) as a new result for \((y, k)\)
- New extended packed node \((x ::= \alpha_i y \cdot \beta_i, l_i, k, r)\) (for all \(i\))
Skip Descend

subject to:
- There are previous results $r_1, \ldots, r_j$ for $(y, k)$

$$
\begin{align*}
(x ::= \alpha \cdot y \beta, l, k) & \rightarrow (x ::= \alpha y \cdot \beta, l, r_1) \\
& \quad \ldots \\
& \rightarrow (x ::= \alpha y \cdot \beta, l, r_j)
\end{align*}
$$

State Update
- Store $(x ::= \alpha y \cdot \beta, l)$ as a continuation for $(y, k)$
- New extended packed node $(x ::= \alpha y \cdot \beta, l, k, r_i)$ (for all $i$)
String “1”, Grammar:

\[e_0 ::= e_1\]
\[e_0 ::= e_0 - e_1\]
\[e_1 ::= INT\]
\[e_1 ::= (e_0)\]

Descriptors

\[(s ::= \cdot e_0, 0, 0),\]

Extended Packed Nodes

Function call | continuations | results
---|---|---
\[(e_0, 0)\] | | 
\[(e_1, 0)\] | |
String “1”, Grammar:

\[
\begin{align*}
e_0 &::= e_1 \\
e_0 &::= e_0 - e_1 \\
e_1 &::= INT \\
e_1 &::= (e_0)
\end{align*}
\]

Descriptors

\[
(s ::= \cdot e_0, 0, 0), \ (e_0 ::= \cdot e_1, 0, 0), \ (e_0 ::= \cdot e_0 - e_1, 0, 0)
\]

Extended Packed Nodes

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
</tr>
</thead>
<tbody>
<tr>
<td>((e_0, 0))</td>
<td>((s ::= e_0\cdot, 0))</td>
</tr>
<tr>
<td>((e_1, 0))</td>
<td></td>
</tr>
</tbody>
</table>
String “1”, Grammar:

\[
\begin{align*}
e_0 & ::= e_1 & e_1 & ::= \text{INT} \\
e_0 & ::= e_0 - e_1 & e_1 & ::= (e_0)
\end{align*}
\]

Descriptors

\[
\begin{align*}
(s & ::= \cdot e_0, 0, 0), (e_0 & ::= \cdot e_1, 0, 0), (e_0 & ::= \cdot e_0 - e_1, 0, 0) \\
(e_1 & ::= \cdot \text{INT}, 0, 0), (e_1 & ::= \cdot (e_0), 0, 0)
\end{align*}
\]

Extended Packed Nodes

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_0, 0)</td>
<td>(s ::= e_0\cdot, 0)</td>
<td></td>
</tr>
<tr>
<td>(e_1, 0)</td>
<td>(e_0 ::= e_1\cdot, 0)</td>
<td></td>
</tr>
</tbody>
</table>
String “1”, Grammar:

\[
e_0 ::= e_1 \\
e_0 ::= e_0 - e_1 \\
e_1 ::= \text{INT} \\
e_1 ::= (e_0)
\]

Descriptors

\[
(s ::= \cdot e_0, 0, 0), (e_0 ::= \cdot e_1, 0, 0), (e_0 ::= \cdot e_0 - e_1, 0, 0) \\
(e_1 ::= \cdot \text{INT}, 0, 0), (e_1 ::= \cdot (e_0), 0, 0), (e_1 ::= \text{INT}\cdot, 0, 1),
\]

Extended Packed Nodes

\[
(e_1 ::= \text{INT}\cdot, 0, 0, 1),
\]

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>((e_0, 0))</td>
<td>((s ::= e_0\cdot, 0))</td>
<td></td>
</tr>
<tr>
<td>((e_1, 0))</td>
<td>((e_0 ::= e_1\cdot, 0))</td>
<td></td>
</tr>
</tbody>
</table>
String “1”, Grammar:

\[
\begin{align*}
e_0 & ::= e_1 \\
e_0 & ::= e_0 - e_1 \\
e_1 & ::= INT \\
e_1 & ::= (e_0)
\end{align*}
\]

Descriptors

\[
\begin{align*}
(s & ::= \cdot e_0, 0, 0), \ (e_0 & ::= \cdot e_1, 0, 0), \ (e_0 & ::= \cdot e_0 - e_1, 0, 0) \\
(e_1 & ::= \cdot INT, 0, 0), \ (e_1 & ::= \cdot (e_0), 0, 0), \ (e_1 & ::= INT \cdot, 0, 1), \\
(e_0 & ::= e_1 \cdot, 0, 1),
\end{align*}
\]

Extended Packed Nodes

\[
\begin{align*}
(e_1 & ::= INT \cdot, 0, 0, 1), \ (e_0 & ::= e_1 \cdot, 0, 0, 1),
\end{align*}
\]

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_0, 0)</td>
<td>(s ::= e_0 \cdot, 0)</td>
<td></td>
</tr>
<tr>
<td>(e_1, 0)</td>
<td>(e_0 ::= e_1 \cdot, 0)</td>
<td>1</td>
</tr>
</tbody>
</table>
String “1”, Grammar:

\[
\begin{align*}
e_0 & ::= e_1 & e_1 & ::= INT \\
e_0 & ::= e_0 - e_1 & e_1 & ::= (e_0)
\end{align*}
\]

Descriptors

\[
\begin{align*}
(s ::= \cdot e_0, 0, 0), & \quad (e_0 ::= \cdot e_1, 0, 0), & (e_0 ::= \cdot e_0 - e_1, 0, 0) \\
(e_1 ::= \cdot INT, 0, 0), & \quad (e_1 ::= \cdot (e_0), 0, 0), & (e_1 ::= INT \cdot, 0, 1), \\
(e_0 ::= e_1 \cdot, 0, 1), & \quad (s ::= e_0 \cdot, 0, 1)
\end{align*}
\]

Extended Packed Nodes

\[
\begin{align*}
(e_1 ::= INT \cdot, 0, 0, 1), & \quad (e_0 ::= e_1 \cdot, 0, 0, 1), & (s ::= e_0 \cdot, 0, 0, 1),
\end{align*}
\]

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_0, 0)</td>
<td>(s ::= e_0 \cdot, 0)</td>
<td>1</td>
</tr>
<tr>
<td>(e_1, 0)</td>
<td>(e_0 ::= e_1 \cdot, 0)</td>
<td>1</td>
</tr>
</tbody>
</table>
String “1”, Grammar:

\[
\begin{align*}
  e_0 & ::= e_1 \\
  e_0 & ::= e_0 - e_1 \\
  e_1 & ::= \text{INT} \\
  e_1 & ::= (e_0)
\end{align*}
\]

Descriptors

\[
\begin{align*}
  (s & ::= \cdot e_0, 0, 0), (e_0 & ::= \cdot e_1, 0, 0), (e_0 & ::= \cdot e_0 - e_1, 0, 0) \\
  (e_1 & ::= \text{INT} \cdot, 0, 0), (e_1 & ::= \cdot (e_0), 0, 0), (e_1 & ::= \text{INT} \cdot, 0, 1), \\
  (e_0 & ::= e_1 \cdot, 0, 1), (s & ::= e_0 \cdot, 0, 1) (e_0 & ::= e_0 \cdot - e_1, 0, 1)
\end{align*}
\]

Extended Packed Nodes

\[
\begin{align*}
  (e_1 & ::= \text{INT} \cdot, 0, 0, 1), (e_0 & ::= e_1 \cdot, 0, 0, 1), (s & ::= e_0 \cdot, 0, 0, 1), \\
  (e_0 & ::= e_0 \cdot - e_1, 0, 0, 1)
\end{align*}
\]

Function call continuations results

<table>
<thead>
<tr>
<th>Function call</th>
<th>continuations</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>((e_0, 0))</td>
<td>((s ::= e_0 \cdot, 0), (e_0 ::= e_0 \cdot - e_1, 0))</td>
<td>1</td>
</tr>
<tr>
<td>((e_1, 0))</td>
<td>((e_0 ::= e_1 \cdot, 0))</td>
<td>1</td>
</tr>
</tbody>
</table>