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 α *or* β

- Does x derive l? Can we keep expanding x till equal to l?

- Terminals: $+ * ( )$ *INT*
- Productions:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= \mathit{INT} \\ e_0 ::= e_1{+}e_0 & e_1 ::= e_2{*}e_1 & e_2 ::= (e_0) \end{array}$$

- Is there a full expansion of e_0 equal to $1{+}2{*}3$?

- Terminals: $+ * ( )$ *INT*
- Productions:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\ e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0) \end{array}$$

- Is there a full expansion of e_0 equal to $1+2*3$?
- 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:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\ e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0) \end{array}$$

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

$\cdot\, e_0$

- Grammar:

$$
\begin{array}{lll}
e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\
e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0)
\end{array}
$$

- Input string: $1+2*3$
- Current index: 0
- Current stack: $[\cdot\, e_0]$

$$e_0 ::= \cdot e_1 + e_0$$

- Grammar:

$$
\begin{array}{lll}
e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\
e_0 ::= e_1{+}e_0 & e_1 ::= e_2{*}e_1 & e_2 ::= (e_0)
\end{array}
$$

- 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:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\ e_0 ::= e_1 {+} e_0 & e_1 ::= e_2 {*} e_1 & e_2 ::= (e_0) \end{array}$$

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

$$e_2 ::= \cdot INT$$

- Grammar:

$$
\begin{array}{lll}
e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\
e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0)
\end{array}
$$

- Input string: 1+2∗3
- Current index: 1
- Current stack: $[e_1 ::= \cdot e_2, e_0 ::= \cdot e_1 + e_0, \cdot e_0]$

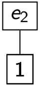
$\boxed{1}$

$e_2 ::= INT \cdot$

- Grammar:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\ e_0 ::= e_1{+}e_0 & e_1 ::= e_2{*}e_1 & e_2 ::= (e_0) \end{array}$$

- Input string: $1{+}2{*}3$
- Current index: 1
- Current stack: $[e_0 ::= \cdot e_1{+}e_0,\ \cdot\ e_0]$

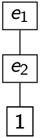


$$e_1 ::= e_2\ \cdot$$

- Grammar:

$$\begin{array}{lll} e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\ e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0) \end{array}$$

- Input string: $1+2*3$
- Current index: 1
- Current stack: $[\cdot\, e_0]$

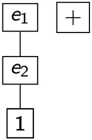


$$e_0 ::= e_1 \cdot + e_0$$

- Grammar:

$$
\begin{array}{lll}
e_0 ::= e_1 & e_1 ::= e_2 & e_2 ::= INT \\
e_0 ::= e_1 + e_0 & e_1 ::= e_2 * e_1 & e_2 ::= (e_0)
\end{array}
$$

- Input string: $1+2*3$
- Current index: 2
- Current stack: $[\cdot\, e_0]$



$$e_0 ::= e_1 + \cdot\, e_0$$

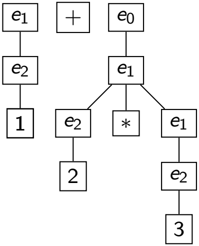
- Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= e_2 \qquad e_2 ::= INT$$

$$e_0 ::= e_1 + e_0 \qquad e_1 ::= e_2 * e_1 \qquad e_2 ::= (e_0)$$

- Input string: 1+2∗3
- Current index: 5
- Current stack: $[\cdot e_0]$

$$e_0 ::= e_1 + e_0 \cdot$$



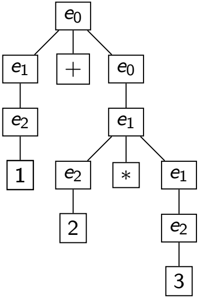
- Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= e_2 \qquad e_2 ::= INT$$

$$e_0 ::= e_1 + e_0 \qquad e_1 ::= e_2 * e_1 \qquad e_2 ::= (e_0)$$

- Input string: 1+2∗3
- Current index: 5
- Current stack: []

$e_0 \cdot$

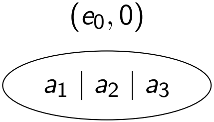


Section 2

Conventional Parser Combinators

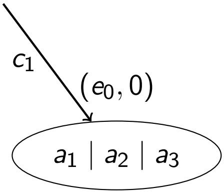
Section 3

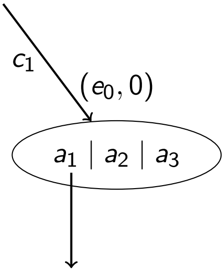
Generalised Parsing



$(e_0, 0)$

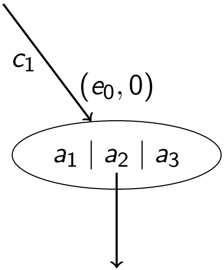
$a_1 \mid a_2 \mid a_3$


c_1
$(e_0, 0)$
$a_1 \mid a_2 \mid a_3$



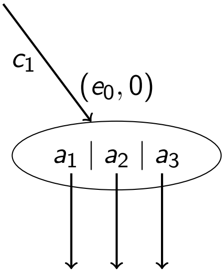
Choosing between alternatives

1. Pick an alternative and stick with it



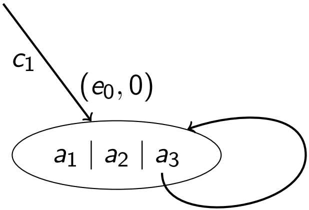
Choosing between alternatives

1. Pick an alternative and stick with it
2. Backtrack to the first alternative that succeeds



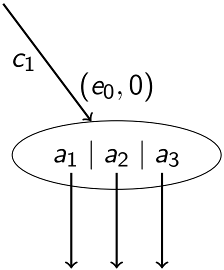
Choosing between alternatives

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 (Scott & Johnstone 2010/2013/2016 - Johnson 1995)

GLL algorithm - Generalised RD parsing

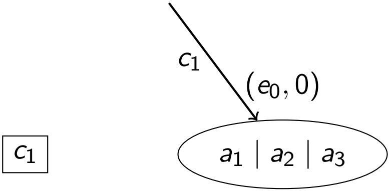
- Explore all alternatives
- No parse function is executed twice with the same arguments
- Can be implemented in $O(n^3)$
- Computes an efficient representation of all parse trees

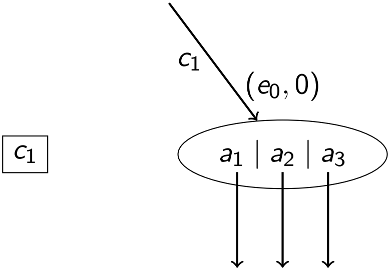
Johnson's memoisation

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

$$(e_0, 0)$$

$$a_1 \mid a_2 \mid a_3$$


c_1
c_1
$(e_0, 0)$
$a_1 \mid a_2 \mid a_3$

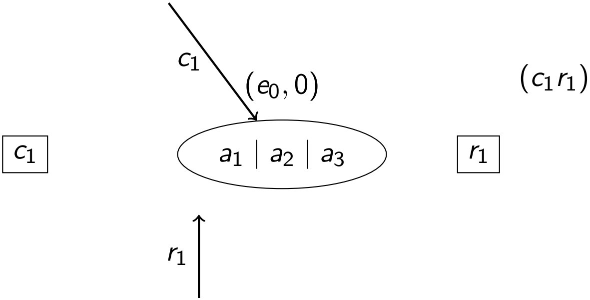


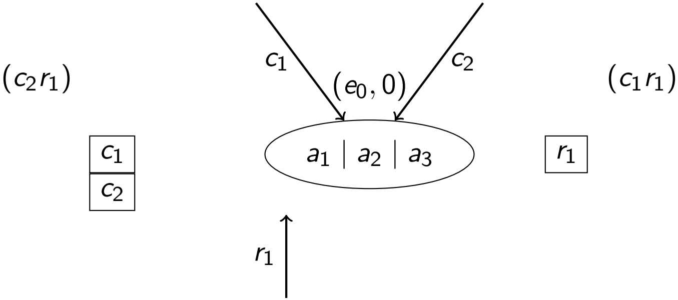
$\boxed{c_1}$

c_1

$(e_0, 0)$

$a_1 \mid a_2 \mid a_3$


c_1
$(e_0, 0)$
$(c_1 r_1)$
c_1
$a_1 \mid a_2 \mid a_3$
r_1
r_1



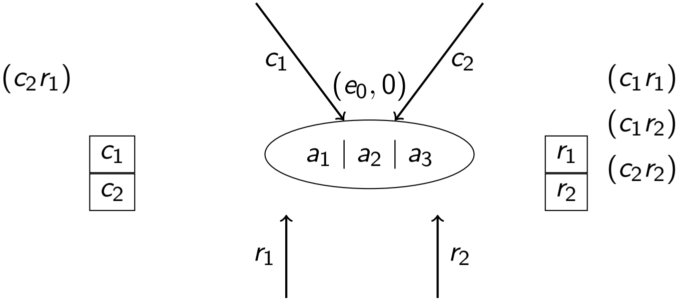
c_1 c_2

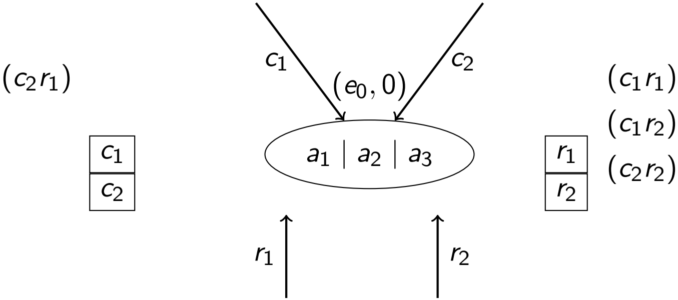
$(c_2 r_1)$ $(e_0, 0)$ $(c_1 r_1)$

$\boxed{c_1}$ $a_1 \mid a_2 \mid a_3$ $\boxed{r_1}$

$\boxed{c_2}$

r_1


$(c_2 r_1)$
c_1
c_2
$(e_0, 0)$
c_1
c_2
$a_1 \mid a_2 \mid a_3$
r_1
r_2
r_1
r_2
$(c_1 r_1)$
$(c_1 r_2)$
$(c_2 r_2)$



- 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

Extended packed nodes

- 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

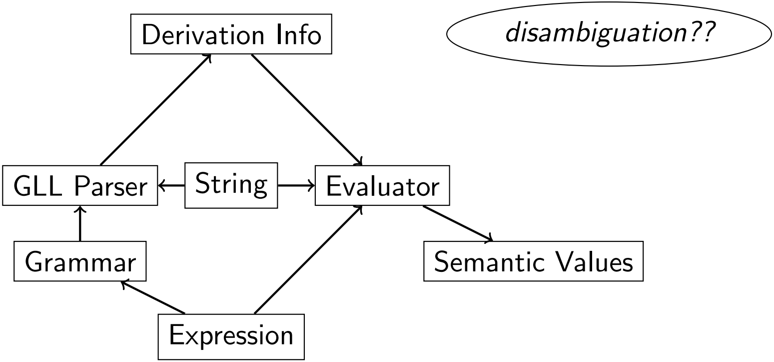
- Simple pure 'solution': Ask the programmer!
- The way we obtain observable sharing influences how derived combinators are defined

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)

GLL Library Overview



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)

```
multiple :: BNF t a → BNF t [a]
multiple x =
   let fresh = mkNt x "*"
   in fresh ⟨::=⟩ (:) ⟨$$⟩ x ⟨**⟩ multiple x    -- x ::= x x*
            ⟨||⟩ satisfy []                     -- x ::= #
```

Combinators for Generalised Parsing

L. Thomas van Binsbergen

Royal Holloway, University of London

October 6, 2016

- With the `gll`-package you build *grammar expressions*

$$
\begin{array}{lll}
\textbf{type}\ \mathit{SymbExpr}\ t\ a & = & \ldots \\
\textbf{type}\ \mathit{AltExpr}\ t\ a & = & \ldots \\
\textbf{type}\ \mathit{AltsExpr}\ t\ a & = & \ldots
\end{array}
$$

$$
\begin{array}{lllll}
(\langle ::= \rangle) & :: \mathit{String} & \rightarrow \mathit{AltsExpr}\ t\ a & \rightarrow & \mathit{SymbExpr}\ t\ a \\
\mathit{term} & :: t & & \rightarrow & \mathit{SymbExpr}\ t\ t \\
(\langle \$\$ \rangle) & :: (a \rightarrow b) & \rightarrow \mathit{SymbExpr}\ t\ a & \rightarrow & \mathit{AltExpr}\ t\ b \\
(\langle {*}{*} \rangle) & :: \mathit{AltExpr}\ t\ (a \rightarrow b) & \rightarrow \mathit{SymbExpr}\ t\ a & \rightarrow & \mathit{AltExpr}\ t\ b \\
\mathit{empty} & :: & & & \mathit{AltsExpr}\ t\ a \\
(\langle || \rangle) & :: \mathit{AltExpr}\ t\ a & \rightarrow \mathit{AltsExpr}\ t\ a & \rightarrow & \mathit{AltsExpr}\ t\ a
\end{array}
$$

- With the `gll`-package you build *grammar expressions*

$$\begin{array}{ll}
\textbf{type}\ \mathit{SymbExpr}\ t\ a & = \ldots \\
\textbf{type}\ \mathit{AltExpr}\ t\ a & = \ldots \\
\textbf{type}\ \mathit{AltsExpr}\ t\ a & = \ldots
\end{array}$$

$$\begin{array}{lllll}
\multicolumn{5}{l}{(\mathit{IsAltExpr}\ \mathit{alt}, \mathit{HasAlts}\ \mathit{alts}, \mathit{IsSymbExpr}\ s) \Rightarrow} \\
(\langle{::=}\rangle) & :: \mathit{String} & \rightarrow \mathit{alts}\ t\ a & \rightarrow & \mathit{SymbExpr}\ t\ a \\
\mathit{term} & :: t & & \rightarrow & \mathit{SymbExpr}\ t\ t \\
(\langle\$\$\rangle) & :: (a \rightarrow b) & \rightarrow s\ t\ a & \rightarrow & \mathit{AltExpr}\ t\ b \\
(\langle{*}{*}\rangle) & :: \mathit{alt}\ t\ (a \rightarrow b) & \rightarrow s\ t\ a & \rightarrow & \mathit{AltExpr}\ t\ b \\
\mathit{empty} & :: & & & \mathit{AltsExpr}\ t\ a \\
(\langle||\rangle) & :: \mathit{alt}\ t\ a & \rightarrow \mathit{alts}\ t\ a & \rightarrow & \mathit{AltsExpr}\ t\ a
\end{array}$$

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

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

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 (for all i)

$$\begin{aligned}(x ::= \alpha \cdot y\beta, l, k) &\rightarrow (y ::= \gamma_1, k, k)\\ &\dots\\ &\rightarrow (y ::= \gamma_i, k, k)\end{aligned}$$

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)

$$\begin{array}{ccc} (y ::= \gamma\cdot, k, r) & \to & (x ::= \alpha_1 y \cdot \beta_1, l_1, r) \\ & \dots & \\ & \to & (x ::= \alpha_i y \cdot \beta_i, l_j, r) \end{array}$$

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)

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

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:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$

$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$,

Extended Packed Nodes

Function call	continuations	results
$(e_0, 0)$ $(e_1, 0)$		

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$
$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

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

Extended Packed Nodes

Function call	continuations	results
$(e_0, 0)$ $(e_1, 0)$	$(s ::= e_0 \cdot, 0)$	

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$

$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$, $(e_0 ::= \cdot e_1, 0, 0)$, $(e_0 ::= \cdot e_0 - e_1, 0, 0)$
$(e_1 ::= \cdot INT, 0, 0)$, $(e_1 ::= \cdot (e_0), 0, 0)$,

Extended Packed Nodes

Function call	continuations	results
$(e_0, 0)$	$(s ::= e_0 \cdot, 0)$	
$(e_1, 0)$	$(e_0 ::= e_1 \cdot, 0)$	

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$

$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$, $(e_0 ::= \cdot e_1, 0, 0)$, $(e_0 ::= \cdot e_0 - e_1, 0, 0)$
$(e_1 ::= \cdot INT, 0, 0)$, $(e_1 ::= \cdot (e_0), 0, 0)$, $(e_1 ::= INT \cdot, 0, 1)$,

Extended Packed Nodes

$(e_1 ::= INT \cdot, 0, 0, 1)$,

Function call	continuations	results
$(e_0, 0)$	$(s ::= e_0 \cdot, 0)$	
$(e_1, 0)$	$(e_0 ::= e_1 \cdot, 0)$	

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$
$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$, $(e_0 ::= \cdot e_1, 0, 0)$, $(e_0 ::= \cdot e_0 - e_1, 0, 0)$
$(e_1 ::= \cdot INT, 0, 0)$, $(e_1 ::= \cdot (e_0), 0, 0)$, $(e_1 ::= INT \cdot, 0, 1)$,
$(e_0 ::= e_1 \cdot, 0, 1)$,

Extended Packed Nodes

$(e_1 ::= INT \cdot, 0, 0, 1)$, $(e_0 ::= e_1 \cdot, 0, 0, 1)$,

Function call	continuations	results
$(e_0, 0)$	$(s ::= e_0 \cdot, 0)$	1
$(e_1, 0)$	$(e_0 ::= e_1 \cdot, 0)$	

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= \mathit{INT}$$

$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$, $(e_0 ::= \cdot e_1, 0, 0)$, $(e_0 ::= \cdot e_0 - e_1, 0, 0)$
$(e_1 ::= \cdot \mathit{INT}, 0, 0)$, $(e_1 ::= \cdot (e_0), 0, 0)$, $(e_1 ::= \mathit{INT} \cdot, 0, 1)$,
$(e_0 ::= e_1 \cdot, 0, 1)$, $(s ::= e_0 \cdot, 0, 1)$

Extended Packed Nodes

$(e_1 ::= \mathit{INT} \cdot, 0, 0, 1)$, $(e_0 ::= e_1 \cdot, 0, 0, 1)$, $(s ::= e_0 \cdot, 0, 0, 1)$,

Function call	continuations	results
$(e_0, 0)$	$(s ::= e_0 \cdot, 0)$	1
$(e_1, 0)$	$(e_0 ::= e_1 \cdot, 0)$	1

String “1”, Grammar:

$$e_0 ::= e_1 \qquad e_1 ::= INT$$

$$e_0 ::= e_0 - e_1 \qquad e_1 ::= (e_0)$$

Descriptors

$(s ::= \cdot e_0, 0, 0)$, $(e_0 ::= \cdot e_1, 0, 0)$, $(e_0 ::= \cdot e_0 - e_1, 0, 0)$
$(e_1 ::= \cdot INT, 0, 0)$, $(e_1 ::= \cdot(e_0), 0, 0)$, $(e_1 ::= INT \cdot, 0, 1)$,
$(e_0 ::= e_1 \cdot, 0, 1)$, $(s ::= e_0 \cdot, 0, 1)$ $(e_0 ::= e_0 \cdot - e_1, 0, 1)$

Extended Packed Nodes

$(e_1 ::= INT \cdot, 0, 0, 1)$, $(e_0 ::= e_1 \cdot, 0, 0, 1)$, $(s ::= e_0 \cdot, 0, 0, 1)$,
$(e_0 ::= e_0 \cdot - e_1, 0, 0, 1)$

Function call	continuations	results
$(e_0, 0)$	$(s ::= e_0 \cdot, 0), (e_0 ::= e_0 \cdot - e_1, 0)$	1
$(e_1, 0)$	$(e_0 ::= e_1 \cdot, 0)$	1