Derivation of a Typed Functional LR Parser

Ralf Hinze (ralf@informatik.uni-bonn.de)
Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany.

Ross Paterson (ross@soi.city.ac.uk)
Department of Computing, City University, London EC1V 0HB, United Kingdom.

Abstract. This paper describes a purely functional implementation of LR parsing. We formally derive our parsers in a series of steps starting from the inverse of printing. In contrast to traditional implementations of LR parsing, the resulting parsers are fully typed, stackless and table-free. The parsing functions pursue alternatives in parallel with each alternative represented by a continuation argument. The direct implementation presents many opportunities for optimization and initial measurements show excellent performance in comparison with conventional table-driven parsers.

Keywords: continuations, derivation, LL- and LR parsing, efficiency.

1. Introduction

Variants of LR parsing (Knu65) have long been the favoured technique for mechanically generated parsers. LR grammars are expressive, allowing a convenient description of most common programming language constructs, and many tools exist to generate efficient parsers from them.

Typical implementations reflect the origin of these parsers as deterministic pushdown automata: there is a table of actions for each state and input symbol, and a stack holding saved values and states. In this paper, we shall derive an alternative implementation: the parsers developed here consist of mutually recursive functions, each taking a number of continuation functions. To give an idea of our destination, Figure 1 lists a complete parser for a tiny expression language. The states of conventional parsers have become functions with continuation arguments, and the stack has disappeared: values discovered during parsing are immediately passed to the continuations. Another salient feature of our parsers is that they are fully typed, without the need for a union type for stack elements as in many other implementations.

We shall explain our parsers by deriving them in a series of program transformation steps. After introducing the notations we shall be using in Section 2, the development is as follows:

– In Section 3, we begin with a direct specification of parsing, as the inverse of flattening a parse tree. Parsing is therefore a non-deterministic, or set-valued, function. It is convenient to work with

```
data Expr          = Ident String | Apply Expr Expr
reduce_id          :: (Expr → Parser r) → (String → Parser r)
reduce_id g n      = g (Ident n)
reduce_E(E)        :: (Expr → Parser r) → (Expr → Expr → Parser r)
reduce_E(E) g f x  = g (Apply f x)
data Token         = IDENT String | LPAREN | RPAREN | EOF
type Parser r      = [Token] → Maybe r
parse_E            :: Parser Expr
parse_E            = state_1 (λe tr → return e)
state_1            :: (Expr → Parser r) → (Parser r)
state_1 k ts       = case ts of IDENT n : tr → reduce_id g n tr
                                _            → fail "syntax error"
  where g e        = state_2 (reduce_E(E) g e) (k e)
state_2            :: (Expr → Parser r) → (Parser r) → (Parser r)
state_2 k_1 k_2 ts = case ts of LPAREN : tr → state_3 k_1 tr
                                EOF : tr    → k_2 tr
                                _           → fail "syntax error"
state_3            :: (Expr → Parser r) → (Parser r)
state_3 k ts       = case ts of IDENT n : tr → reduce_id g n tr
                                _            → fail "syntax error"
  where g e        = state_4 (reduce_E(E) g e) (k e)
state_4            :: (Expr → Parser r) → (Parser r) → (Parser r)
state_4 k_1 k_2 ts = case ts of LPAREN : tr → state_3 k_1 tr
                                RPAREN : tr → k_2 tr
                                _           → fail "syntax error"
```

Figure 1. Parser for the grammar $E \to$ `id` | E `(` E `)`.

such functions through most of our development, addressing the deterministic special case towards the end.

– The next step, in Section 4, is to derive a simple recursive characterization of the parsing functions. This form corresponds to continuation-style recursive descent parsers, and is thus directly executable only if the grammar is not left-recursive.

– Left recursion is removed in Section 5 by analysing these functions as sets of sequences of primitive steps, given by a left-linear grammar, and using the standard technique to obtain an equivalent right-linear grammar. The resulting program corresponds to a non-deterministic LR parser.

– Finally, in Section 6, we remove non-determinism by collecting sets of parsing functions, so that they parse alternatives in paral-

lel, obtaining deterministic LR parsers. We also consider various optimizations. Some of these have counterparts for conventional parsers; some are specific to the functional representation.

We discuss previous derivations of bottom-up functional parsers in Section 7, and compare them with the parsers derived here.

Experiments with a range of grammars, reported in Section 8, show that our parsers compare well to conventional table-based ones.

Finally, Section 9 concludes.

2. Notation

This section briefly introduces the notation used in the subsequent sections. The reader may wish to skim through the material and come back later for reference.

Sets As noted in the introduction we shall derive an implementation of LR parsing in a series of steps, each of which yields an executable program. For concreteness, the derivation and the resulting programs are presented in a Haskell-like notation; for information on Haskell, see (Pey03). Although we use Haskell as a vehicle, we are interpreting it in the category of sets and total functions so that arbitrary recursion is unavailable. At the end of the development, when all non-determinism is removed, we shall switch to the conventional interpretation of Haskell in the category of complete partial orders and continuous functions.

Furthermore, we shall make heavy use of set comprehension notation. As an example, $\mathsf{prefixes}$ is a function that yields the set of all prefixes of a given list.

$$\begin{aligned}&\mathsf{prefixes} \quad :: [\mathfrak{a}] \to \mathbb{P}\,[\mathfrak{a}]\\ &\mathsf{prefixes}\ \mathsf{x} = \{\mathsf{p} \mid \mathsf{p} +\!\!+ \mathsf{s} = \mathsf{x}\}\end{aligned}$$

Here $[\mathfrak{a}]$ is the type of lists of elements of type $\mathfrak{a}$; likewise, $\mathbb{P}\ \mathfrak{a}$ is the type of sets of elements of type $\mathfrak{a}$; '$+\!\!+$' denotes list concatenation. Note that the variables p and s in the body of the comprehension are implicitly existentially quantified so that the definition is shorthand for the more verbose

$$\mathsf{prefixes}\ \mathsf{x} = \{\mathsf{p} \mid \exists\, \mathsf{p} :: [\mathfrak{a}]\,.\, \exists\, \mathsf{s} :: [\mathfrak{a}]\,.\, \mathsf{p} +\!\!+ \mathsf{s} = \mathsf{x}\}$$

For the derivation we prefer the shorter form, so to reduce clutter we adopt the convention of (Hin02) that the variables to the left of '$=$' and '$\in$' are always implicitly existentially quantified.

Context-free grammars We shall use standard notation for context-free grammars with one minor exception: we allow a set of start symbols. Formally, a *context-free grammar* $(\mathbf{T}, \mathbf{N}, \mathbf{S}, \mathbf{P})$ consists of

- disjoint sets $\mathbf{T}$ and $\mathbf{N}$, called respectively *terminal* and *nonterminal symbols*, together comprising the set $\mathbf{V} = \mathbf{T} \cup \mathbf{N}$ of *symbols*,
- a set $\mathbf{S} \subseteq \mathbf{N}$ of *start symbols*, and
- a set $\mathbf{P} \subseteq \mathbf{N} \times \mathbf{V}^*$ of *productions.*

We shall use the following conventions for elements of these sets:

$$\begin{array}{llll} a, b, c, d & \in \mathbf{T} & w & \in \mathbf{T}^* \\ A, B, C, D & \in \mathbf{N} & & \\ X & \in \mathbf{V} & \alpha, \beta, \gamma & \in \mathbf{V}^* \end{array}$$

Elements of $\mathbf{P}$ are written $A \to \alpha$.

Example 1. Consider the simple expression grammar (ASU86, p222) with

$$\begin{array}{lcl} \mathbf{T} & = & \{\, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id} \,\} \\ \mathbf{N} & = & \{\, E, T, F \,\} \\ \mathbf{S} & = & \{\, E \,\} \end{array}$$

and the following productions:

$$\begin{array}{lcl} E & \to & E \texttt{ + } T \\ E & \to & T \\ T & \to & T \texttt{ * } F \\ T & \to & F \\ F & \to & \texttt{(} E \texttt{)} \\ F & \to & \texttt{id} \end{array}$$

We shall use this grammar as a running example. □

For any grammar, we can define an *augmented grammar* (we tacitly assume that $\$ \notin \mathbf{V}$):

$$\begin{array}{lcl} \mathbf{T}^\dagger & = & \mathbf{T} \cup \{\, \$ \,\} \\ \mathbf{N}^\dagger & = & \mathbf{N} \cup \mathbf{S}^\dagger \\ \mathbf{S}^\dagger & = & \{\, S^\dagger \mid S \in \mathbf{S} \,\} \\ \mathbf{P}^\dagger & = & \mathbf{P} \cup \{\, S^\dagger \to S\$ \mid S \in \mathbf{S} \,\} \end{array}$$

In an augmented grammar the start symbols do not appear on the right-hand side of an production. Furthermore, the end of input is signalled by the special symbol '$'.

Linear grammars A *left-linear grammar* is one in which all productions have the form $A \to Bw$ or $A \to w$. Similarly, a *right-linear grammar* has only productions of the forms $A \to wB$ and $A \to w$.

Any left-linear grammar $(\mathbf{T}, \{S\} \uplus \mathbf{N}, \{S\}, \mathbf{P})$ with a single start symbol S that does not appear on the right-hand side of a production may be transformed into an equivalent right-linear grammar $(\mathbf{T}, \{S\} \uplus \bar{\mathbf{N}}, \{S\}, \bar{\mathbf{P}})$, or *vice versa*, by the rules:

$$\begin{array}{lllll} \mathbf{P} \ni & S \to Aw & \leftrightsquigarrow & \bar{A} \to w & \in \bar{\mathbf{P}} \\ & S \to w & \leftrightsquigarrow & S \to w & \\ & A \to Bw & \leftrightsquigarrow & \bar{B} \to w\bar{A} & \\ & A \to w & \leftrightsquigarrow & S \to w\bar{A} & \end{array}$$

Items An *item* is a production plus a position in the right-hand side, written $A \to \alpha \cdot \beta$ where the dot marks the position. *Kernel items* of an augmented grammar $(\mathbf{T}^\dagger, \mathbf{N}^\dagger, \mathbf{S}^\dagger, \mathbf{P}^\dagger)$ are those of the forms $S^\dagger \to \cdot\, S\$$ or $A \to \alpha \cdot \beta$ with $\alpha \neq \epsilon$. We let i and j denote arbitrary items, and use q for kernel items.

We also define a predict relation '$\triangleright$' on items by $(A \to \alpha \cdot C\beta) \triangleright (C \to \cdot\, \gamma)$, where $C \to \gamma \in \mathbf{P}^\dagger$, with transitive closure $\triangleright^+$ and reflexive-transitive closure $\triangleright^*$. Furthermore, let $q^+ = \{\, i \mid q \triangleright^+ i \,\}$ and $q^* = \{\, i \mid q \triangleright^* i \,\}$. Note that $q^+ = q^* \setminus \{\, q \,\}$ if and only if q is a kernel item. For example, the equation does not hold for $A \to \cdot\, Aa$.

Finally, for any set Q of items, we define

$$\begin{array}{ll} Q/X & = \{\, A \to \alpha \cdot X\beta \in Q \,\} \\ Q{+}X & = \{\, A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in Q \,\} \\ \mathit{next}(Q) & = \{\, X \mid A \to \alpha \cdot X\beta \in Q \,\} \end{array}$$

3. A specification of parsing

3.1. Attributes

In most parsing applications, we wish to collect information about the string parsed. The standard approach is to define for each symbol X a sequence of data types $\bar{X}$ of *attributes* computed for each parse of the symbol. (We consider only synthesized attributes here.) It would suffice to have a single attribute type for each symbol, representing multiple attributes by tuples and no attributes by a unit type. However, we choose to retain the sequences of types because they are more efficient, and do not significantly complicate the description as we will need to

deal with sequences of types anyway. We adopt the notational device of extending the function type to sequences of types, allowing

$$\mathsf{t}_1 \ \cdots \ \mathsf{t}_n \to \mathsf{t}$$

as an alternative form of

$$\mathsf{t}_1 \to \cdots \to \mathsf{t}_n \to \mathsf{t}$$

Once we have defined $\bar{X}$ for each symbol X, we can form $\bar{\alpha}$ by concatenating these sequences of types for each symbol in α. We shall also write ν_α for a sequence of variables with types $\bar{\alpha}$.

3.2. Tokens

We assume for each terminal symbol a, a string of attribute types $\bar{a}$, with $\bar{\$} = \epsilon$. The combination of a terminal symbol with attribute values is conventionally called a *token* (AU72). We can represent these with a Haskell data type as follows:

$$\mathbf{data}\ \mathsf{Token} \ = \ \textstyle\sum_{c \in \mathbf{T}^\dagger} \mathsf{T}_c \ \bar{c}$$

For example, a Token type and a suitable lexical analyser for the expression grammar, Example 1, are given in Figure 2. In this case, some symbols have a single attribute, while others have none. Note that '+' and '∗' have been generalized to additive and multiplicative operator families whose attribute specifies the concrete operator at hand. The token EOF corresponds to $\mathsf{T}_\$$.

3.3. Parse trees

In general, productions would be annotated with rules for computing the attributes of the left side from those of the right side. Our aim here, though, is to derive parsing as an invertible process, so we shall use parse trees as attributes, thus ensuring that no information about the parse is lost. Hence we define for each nonterminal symbol A, a parse tree type $\bar{A}$ as follows

$$\mathbf{data}\ \bar{A} \ = \ \textstyle\sum_{(A \to \alpha) \in \mathbf{P}} \mathsf{Rule}_{A \to \alpha} \ \bar{\alpha}$$

Note that $\mathsf{Rule}_{A \to \alpha}$ has type $\bar{\alpha} \to \bar{A}$ demonstrating that the parse tree constructor is the inverse of a production. As an exception, in the case of the start symbols $S^\dagger$ of an augmented grammar, we define $\overline{S^\dagger} = \bar{S}$. For example, parse tree types for the expression grammar appear in Figure 3.

Parse trees are an initial algebra of the signature determined by the grammar. The attributes that are usually associated with nonterminal

```
data Token  = IDENT  String
            | ADDOP AddOp                -- '+' and '-'
            | MULOP MulOp                -- '*' and '/'
            | LPAREN
            | RPAREN
            | EOF                        -- = T$
data AddOp = Plus  | Minus
data MulOp = Times | Divide

tokens           :: String → [Token]
tokens []        = [EOF]
tokens ('+' : cs) = ADDOP Plus   : tokens cs
tokens ('-' : cs) = ADDOP Minus  : tokens cs
tokens ('*' : cs) = MULOP Times  : tokens cs
tokens ('/' : cs) = MULOP Divide : tokens cs
tokens ('(' : cs) = LPAREN       : tokens cs
tokens (')' : cs) = RPAREN       : tokens cs
tokens (c   : cs)
  | isAlpha c    = IDENT (c : n)  : tokens cs'
  | otherwise    = tokens cs
  where (n, cs') = span isAlphaNum cs
```

Figure 2. Tokens for the expression grammar.

```
data Expr   = Rule_{E→E+T} Expr AddOp Term
            | Rule_{E→T}   Term
data Term   = Rule_{T→T*F} Term MulOp Factor
            | Rule_{T→F}   Factor
data Factor = Rule_{F→(E)} Expr
            | Rule_{F→id}  String
```

Figure 3. Parse tree types for the expression grammar.

symbols may be computed by a fold over the parse tree. This fold may be automatically fused with the parser by a routine application of deforestation (Wad88), replacing the constructors $\mathsf{Rule}_{A\to\alpha}$ with functions that compute the values of the attributes (see also Section 6.4).

```
class ParseTree a where
  flatten                        :: a → [Token]
instance ParseTree Expr where
  flatten (Rule_{E→E+T} e op t) = flatten e ++ [ADDOP op] ++ flatten t
  flatten (Rule_{E→T}   t)      = flatten t
instance ParseTree Term where
  flatten (Rule_{T→T*F} t op f) = flatten t ++ [MULOP op] ++ flatten f
  flatten (Rule_{T→F}   f)      = flatten f
instance ParseTree Factor where
  flatten (Rule_{F→(E)} e)      = [LPAREN] ++ flatten e ++ [RPAREN]
  flatten (Rule_{F→id}  n)      = [IDENT n]
```

Figure 4. Flattening parse trees for the expression grammar.

As a special case of this fold, we can define a family of functions $\mathtt{flatten}_\alpha$ for each string α, with which we can recover the original sequence of tokens from a parse tree:

$$
\begin{array}{lcl}
\mathtt{flatten}_\alpha & :: & \bar{\alpha} \to [\mathsf{Token}] \\
\mathtt{flatten}_{S^\dagger}\ \nu & = & \mathtt{flatten}_{S\$}\ \nu \\
\mathtt{flatten}_a\ \nu_a & = & [\mathsf{T}_a\ \nu_a] \\
\mathtt{flatten}_A\ (\mathsf{Rule}_{A\to\alpha}\ \nu_\alpha) & = & \mathtt{flatten}_\alpha\ \nu_\alpha \\
\mathtt{flatten}_\epsilon & = & [\,] \\
\mathtt{flatten}_{\alpha\beta}\ \nu_\alpha\ \nu_\beta & = & \mathtt{flatten}_\alpha\ \nu_\alpha +\!\!+ \mathtt{flatten}_\beta\ \nu_\beta
\end{array}
$$

This family of functions, together with the parse tree types, thus encode the grammar in Haskell. The initiality of parse trees captures the notion that the language defined is the least fixed point of the grammar in the subset ordering. Suitable definitions for the expression grammar are given in Figure 4. Note that the family of functions is represented by an overloaded function in Haskell.

3.4. Non-deterministic parsing

Parsing can now be described as an inverse of the function $\mathtt{flatten}_{S^\dagger}$. Of course, this function is not necessarily surjective or injective: some token strings may not be the flattening of any parse tree, while others may correspond to more than one parse tree, if the grammar is ambiguous. Although for practical applications we are primarily interested in unambiguous grammars, specifically LALR ones, it will be cleaner to develop our parsers for the general case. In general, a token string may be the flattening of a number of parse trees (possibly zero), so we define

```
type Parser r = [Token] → ℙ r
```

Specification of parse The parse function for a start symbol S will have type

$$\mathsf{parse}_S \;::\; \mathsf{Parser}\;\bar{S}$$

We shall expect it to satisfy

$$\mathsf{parse}_S\;\mathsf{ts} \;=\; \{\,\mathsf{v}_S \mid \mathsf{flatten}_{S^\dagger}\;\mathsf{v}_S +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \tag{1}$$

Following the conventions discussed in Section 2, the variables v_S and tr are implicitly bound in the set comprehension. In this case, the suffix tr is actually immaterial, since $\mathsf{flatten}_{S^\dagger}\;\mathsf{v}_S$ will end with the special token $\mathsf{T}_\$$ (EOF in the expression parser). Anything after that token will be ignored by the parser.

4. Non-deterministic recursive descent parsing

We cannot expect a recursive definition of parse_S. Rather we shall define it in terms of a more general auxiliary function that parses a suffix of a production. Following convention we use items of the form $A \to \alpha \cdot \beta$ to indicate this position. In the sequel we specify a parser for each item $A \to \alpha \cdot \beta$, called $\mathsf{state}_{A\to\alpha\cdot\beta}$, with which we can define parse_S. We derive equations that this family of functions satisfies, and show that it is the least such family. We thus obtain a recursive characterization, which corresponds to recursive descent parsing.

Specification of state Suppose that during the parse we are at a point described by an item $A \to \alpha \cdot \beta$. We expect the remaining input to be parsable as a β, followed by further tokens. Hence the function to parse from this point is passed a parsing continuation, to be used after the β is seen. In parsing the string β, our parser will obtain attribute values of type $\bar{\beta}$, and these will be passed to the continuation. Hence the parsing function corresponding to $A \to \alpha \cdot \beta$ has the signature

$$\mathsf{state}_{A\to\alpha\cdot\beta} \;::\; (\bar{\beta} \to \mathsf{Parser}\;\mathsf{r}) \to \mathsf{Parser}\;\mathsf{r}$$

and we shall require that it satisfies

$$\mathsf{state}_{A\to\alpha\cdot\beta}\;\mathsf{k}\;\mathsf{ts} \;=\; \bigcup\{\,\mathsf{k}\;\mathsf{v}_\beta\;\mathsf{tr} \mid \mathsf{flatten}_\beta\;\mathsf{v}_\beta +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \tag{2}$$

Note that $\mathsf{state}_{A\to\alpha\cdot\beta}$ is polymorphic in the result type r, which allows us to support parsers with multiple entry points (corresponding to grammars with multiple start symbols). Note further that $\mathsf{state}_{A\to\alpha\cdot\beta}$ neither depends on A nor on α. Consequently, the state functions are

identical if the strings to the right of the dot are identical, that is, $\mathtt{state}_{A \to \alpha \cdot \beta} = \mathtt{state}_{A' \to \alpha' \cdot \beta}$. We return to this issue in Section 6.3.

Derivation of parse Parsing begins with the item $S^\dagger \to \cdot\, S\$$:

$$
\begin{array}{ll}
 & \mathtt{parse}_S\ \mathtt{ts} \\
= & \quad \langle\ \text{specification of } \mathtt{parse}\ \rangle \\
 & \{\, v_S \mid \mathtt{flatten}_{S^\dagger}\ v_S +\!\!+ \mathtt{tr} = \mathtt{ts} \,\} \\
= & \quad \langle\ \text{definition of } \mathtt{flatten}\ \rangle \\
 & \{\, v_S \mid \mathtt{flatten}_{S\$}\ v_S +\!\!+ \mathtt{tr} = \mathtt{ts} \,\} \\
= & \quad \langle\ \text{sets}\ \rangle \\
 & \bigcup \{\, \{\, v_S \,\} \mid \mathtt{flatten}_{S\$}\ v_S +\!\!+ \mathtt{tr} = \mathtt{ts} \,\} \\
= & \quad \langle\ \text{introduce } \mathtt{accept} \text{—see below}\ \rangle \\
 & \bigcup \{\, \mathtt{accept}\ v_S\ \mathtt{tr} \mid \mathtt{flatten}_{S\$}\ v_S +\!\!+ \mathtt{tr} = \mathtt{ts} \,\} \\
= & \quad \langle\ \text{specification of } \mathtt{state} \text{ and } \overline{S\$} = \bar{S}\ \rangle \\
 & \mathtt{state}_{S^\dagger \to \cdot\, S\$}\ \mathtt{accept}\ \mathtt{ts}
\end{array}
$$

where the function $\mathtt{accept}$ is defined as

$$
\begin{array}{l}
\mathtt{accept} \quad :: \mathtt{r} \to \mathtt{Parser\ r} \\
\mathtt{accept}\ v\ \mathtt{tr} = \{\, v \,\}
\end{array}
$$

and so we have

$$\mathtt{parse}_S \;=\; \mathtt{state}_{S^\dagger \to \cdot\, S\$}\ \mathtt{accept}$$

Derivation of state We proceed by case analysis of the string to the right of the dot.

Case $A \to \alpha \cdot$: Consider first items with an empty string to the right of the dot. In this case, we can use the parsing continuation to parse the rest of $\mathtt{ts}$:

$$
\begin{array}{ll}
 & \mathtt{state}_{A \to \alpha \cdot}\ \mathtt{k}\ \mathtt{ts} \\
= & \quad \langle\ \text{specification of } \mathtt{state}\ \rangle \\
 & \bigcup \{\, \mathtt{k}\ v_\epsilon\ \mathtt{tr} \mid \mathtt{flatten}_\epsilon\ v_\epsilon +\!\!+ \mathtt{tr} = \mathtt{ts} \,\} \\
= & \quad \langle\ \text{definition of } \mathtt{flatten}\ \rangle \\
 & \bigcup \{\, \mathtt{k}\ \mathtt{ts} \,\} \\
= & \quad \langle\ \text{sets}\ \rangle \\
 & \mathtt{k}\ \mathtt{ts}
\end{array}
$$

In a conventional setting this corresponds to a reduce step.

Case $A \to \alpha \cdot X\beta$: If there is at least one symbol to the right of the dot, we have

$$
\begin{array}{ll}
& \mathsf{state}_{A\to\alpha\cdot X\beta}\ \mathsf{k}\ \mathsf{ts} \\
= & \quad \langle\ \text{specification of } \mathsf{state}\ \rangle \\
& \bigcup\{\, \mathsf{k}\ v_{X\beta}\ \mathsf{tr} \mid \mathsf{flatten}_{X\beta}\ v_{X\beta} +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{definition of } \mathsf{flatten}\ \rangle \\
& \bigcup\{\, \mathsf{k}\ v_X\ v_\beta\ \mathsf{tr} \mid \mathsf{flatten}_X\ v_X +\!\!+ \mathsf{flatten}_\beta\ v_\beta +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{sets}\ \rangle \\
& \bigcup\{\, \bigcup\{\, \mathsf{k}\ v_X\ v_\beta\ \mathsf{tr} \mid \mathsf{flatten}_\beta\ v_\beta +\!\!+ \mathsf{tr} = \mathsf{tr}'\,\} \\
& \quad \mid \mathsf{flatten}_X\ v_X +\!\!+ \mathsf{tr}' = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{specification of } \mathsf{state}\ \rangle \\
& \bigcup\{\, \mathsf{state}_{A\to\alpha X\cdot\beta}\ (\mathsf{k}\ v_X)\ \mathsf{tr}' \mid \mathsf{flatten}_X\ v_X +\!\!+ \mathsf{tr}' = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{introduce } \mathsf{goto}\text{—see below}\ \rangle \\
& \bigcup\{\, \mathsf{goto}_{A\to\alpha\cdot X\beta}\ \mathsf{k}\ v_X\ \mathsf{tr}' \mid \mathsf{flatten}_X\ v_X +\!\!+ \mathsf{tr}' = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{introduce } \mathsf{shift}\text{—see below}\ \rangle \\
& \mathsf{shift}_X\ (\mathsf{goto}_{A\to\alpha\cdot X\beta}\ \mathsf{k})\ \mathsf{ts}
\end{array}
$$

where functions $\mathsf{goto}_{A\to\alpha\cdot X\beta}$ and shift_X are specified by

$$
\begin{array}{lcl}
\mathsf{goto}_{A\to\alpha\cdot X\beta} & :: & (\overline{X\beta} \to \mathsf{Parser\ r}) \to (\bar{X} \to \mathsf{Parser\ r}) \\
\mathsf{goto}_{A\to\alpha\cdot X\beta}\ \mathsf{k}\ v_X & = & \mathsf{state}_{A\to\alpha X\cdot\beta}\ (\mathsf{k}\ v_X) \\
\mathsf{shift}_X & :: & (\bar{X} \to \mathsf{Parser\ r}) \to \mathsf{Parser\ r} \\
\mathsf{shift}_X\ \mathsf{g}\ \mathsf{ts} & = & \bigcup\{\, \mathsf{g}\ v_X\ \mathsf{tr} \mid \mathsf{flatten}_X\ v_X +\!\!+ \mathsf{tr} = \mathsf{ts}\,\}
\end{array}
$$

As further progress depends on the nature of X, we have introduced shift as a name for the branch point. We also introduce goto to cast shift in a more convenient form for further manipulation: $\mathsf{goto}_{A\to\alpha\cdot X\beta}$ feeds attributes to the continuation and enters the state after X.

Derivation of shift We proceed by case analysis on X.

Case $X = c$: When X is a terminal symbol c, we have

$$
\begin{array}{ll}
& \mathsf{shift}_c\ \mathsf{g}\ \mathsf{ts} \\
= & \quad \langle\ \text{specification of } \mathsf{shift}\ \rangle \\
& \bigcup\{\, \mathsf{g}\ v_c\ \mathsf{tr} \mid \mathsf{flatten}_c\ v_c +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{definition of } \mathsf{flatten}\ \rangle \\
& \bigcup\{\, \mathsf{g}\ v_c\ \mathsf{tr} \mid [\mathsf{T}_c\ v_c] +\!\!+ \mathsf{tr} = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{definition of `}+\!\!+\text{'}\ \rangle \\
& \bigcup\{\, \mathsf{g}\ v_c\ \mathsf{tr} \mid \mathsf{T}_c\ v_c : \mathsf{tr} = \mathsf{ts}\,\} \\
= & \quad \langle\ \text{case expression}\ \rangle \\
& \mathbf{case}\ \mathsf{ts}\ \mathbf{of}\ (\mathsf{T}_c\ v_c : \mathsf{tr}) \to \mathsf{g}\ v_c\ \mathsf{tr} \\
& \qquad\qquad\quad\ \ \_ \to \varnothing
\end{array}
$$

In a conventional setting this corresponds to a shift step.

Case $X = C$: When X is a nonterminal symbol C, we have

$$\begin{array}{ll} & \mathsf{shift}_C\ \mathsf{g}\ \mathsf{ts} \\ = & \quad \langle\ \text{specification of } \mathsf{shift}\ \rangle \\ & \bigcup \{\, \mathsf{g}\ v_C\ \mathsf{tr} \mid \mathsf{flatten}_C\ v_C +\!\!+ \mathsf{tr} = \mathsf{ts} \,\} \\ = & \quad \langle\ \text{definition of } \mathsf{flatten}\ \rangle \\ & \bigcup \{\, \mathsf{g}\ (\mathsf{Rule}_{C\to\gamma}\ v_\gamma)\ \mathsf{tr} \mid C \to \gamma \in \mathbf{P} \wedge \mathsf{flatten}_\gamma\ v_\gamma +\!\!+ \mathsf{tr} = \mathsf{ts} \,\} \\ = & \quad \langle\ \text{introduce } \mathsf{reduce}\text{—see below}\ \rangle \\ & \bigcup \{\, \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}\ v_\gamma\ \mathsf{tr} \mid C \to \gamma \in \mathbf{P} \wedge \mathsf{flatten}_\gamma\ v_\gamma +\!\!+ \mathsf{tr} = \mathsf{ts} \,\} \\ = & \quad \langle\ \text{sets}\ \rangle \\ & \bigcup \{\, \bigcup \{\, \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}\ v_\gamma\ \mathsf{tr} \mid \mathsf{flatten}_\gamma\ v_\gamma +\!\!+ \mathsf{tr} = \mathsf{ts} \,\} \\ & \quad \mid C \to \gamma \in \mathbf{P} \,\} \\ = & \quad \langle\ \text{specification of } \mathsf{state}\ \rangle \\ & \bigcup \{\, \mathsf{state}_{C\to\cdot\gamma}\ (\mathsf{reduce}_{C\to\gamma}\ \mathsf{g})\ \mathsf{ts} \mid C \to \gamma \in \mathbf{P} \,\} \end{array}$$

where we define $\mathsf{reduce}_{C\to\gamma}$ as the continuation-passing counterpart of $\mathsf{Rule}_{C\to\gamma}$:

$$\begin{array}{lcl} \mathsf{reduce}_{C\to\gamma} & :: & (\bar{C} \to \mathsf{Parser\ r}) \to (\bar{\gamma} \to \mathsf{Parser\ r}) \\ \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}\ v_\gamma & = & \mathsf{g}\ (\mathsf{Rule}_{C\to\gamma}\ v_\gamma) \end{array}$$

This step, the only remaining source of nondeterminism, has no counterpart in a conventional setting. We shall transform it further in the next section. For the expression parser, the reduce functions are given in Figure 4. Note that each constructor $\mathsf{Rule}_{C\to\gamma}$ is mentioned once only. As remarked in Section 3.3, when the parser is fused with a fold over a parse tree and deforested, these constructors will be replaced by actions.

To summarize, Figure 6 lists the derived properties of state and of the auxiliary functions we have introduced.

In general, state will not be the sole family of functions satisfying these equations, but we can show that it is the least such family under the set inclusion ordering, lifted pointwise to functions. Let state', shift' and goto' be another such family. Then we wish to establish that $\mathsf{state} \subseteq \mathsf{state}'$ and $\mathsf{shift} \subseteq \mathsf{shift}'$, or equivalently

$$\mathsf{k}\ v_\beta\ \mathsf{tr} \ \subseteq\ \mathsf{state}'_{A\to\alpha\,\cdot\,\beta}\ \mathsf{k}\ (\mathsf{flatten}_\beta\ v_\beta +\!\!+ \mathsf{tr}) \tag{3}$$

$$\mathsf{g}\ v_X\ \mathsf{ts} \ \subseteq\ \mathsf{shift}'_X\ \mathsf{g}\ (\mathsf{flatten}_X\ v_X +\!\!+ \mathsf{ts}) \tag{4}$$

The first set inclusion states that state' produces every parse tree by parsing its flattening and possibly others. The proof is similar in structure to the derivation: We show (3) by induction over the depth of the parse forest v_β with (4) serving as an intermediate result. The details of the proof are omitted.

If the grammar is not left-recursive, the equations of Figure 6 are, in fact, executable, acting as a nondeterministic LL(0) parser. The parser

$$
\begin{array}{ll}
\mathsf{reduce}_{E\to E+T} & :: (\mathsf{Expr} \to \mathsf{Parser\ r}) \\
 & \quad \to (\mathsf{Expr} \to \mathsf{AddOp} \to \mathsf{Term} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{E\to E+T}\ \mathsf{g\ e\ op\ t} & = \mathsf{g}\ (\mathsf{Rule}_{E\to E+T}\ \mathsf{e\ op\ t}) \\
\mathsf{reduce}_{E\to T} & :: (\mathsf{Expr} \to \mathsf{Parser\ r}) \to (\mathsf{Term} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{E\to T}\ \mathsf{g\ t} & = \mathsf{g}\ (\mathsf{Rule}_{E\to T}\ \mathsf{t}) \\
\mathsf{reduce}_{T\to T*F} & :: (\mathsf{Term} \to \mathsf{Parser\ r}) \\
 & \quad \to (\mathsf{Term} \to \mathsf{MulOp} \to \mathsf{Factor} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{T\to T*F}\ \mathsf{g\ t\ op\ f} & = \mathsf{g}\ (\mathsf{Rule}_{T\to T*F}\ \mathsf{t\ op\ f}) \\
\mathsf{reduce}_{T\to F} & :: (\mathsf{Term} \to \mathsf{Parser\ r}) \to (\mathsf{Factor} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{T\to F}\ \mathsf{g\ f} & = \mathsf{g}\ (\mathsf{Rule}_{T\to F}\ \mathsf{f}) \\
\mathsf{reduce}_{F\to (\ E\ )} & :: (\mathsf{Factor} \to \mathsf{Parser\ r}) \to (\mathsf{Expr} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{F\to (\ E\ )}\ \mathsf{g\ e} & = \mathsf{g}\ (\mathsf{Rule}_{F\to (\ E\ )}\ \mathsf{e}) \\
\mathsf{reduce}_{F\to \mathtt{id}} & :: (\mathsf{Factor} \to \mathsf{Parser\ r}) \to (\mathsf{String} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{F\to \mathtt{id}}\ \mathsf{g\ n} & = \mathsf{g}\ (\mathsf{Rule}_{F\to \mathtt{id}}\ \mathsf{n})
\end{array}
$$

Figure 5. Reduce functions for the expression grammar.

$$
\begin{array}{lcl}
\mathsf{state}_{A\to\alpha\cdot\beta} & :: & (\bar{\beta} \to \mathsf{Parser\ r}) \to \mathsf{Parser\ r} \\
\mathsf{state}_{A\to\alpha\cdot}\ \mathsf{k} & = & \mathsf{k} \\
\mathsf{state}_{A\to\alpha\cdot X\beta}\ \mathsf{k} & = & \mathsf{shift}_X\ (\mathsf{goto}_{A\to\alpha\cdot X\beta}\ \mathsf{k}) \\
 & & \\
\mathsf{shift}_X & :: & (\bar{X} \to \mathsf{Parser\ r}) \to \mathsf{Parser\ r} \\
\mathsf{shift}_c\ \mathsf{g\ ts} & = & \mathbf{case}\ \mathsf{ts}\ \mathbf{of}\ (\mathsf{T}_c\ v_c : \mathsf{tr}) \to \mathsf{g}\ v_c\ \mathsf{tr} \\
 & & \qquad\qquad\qquad\quad\ \ \_ \to \varnothing \\
\mathsf{shift}_C\ \mathsf{g\ ts} & = & \bigcup \{\, \mathsf{state}_{C\to\cdot\gamma}\ (\mathsf{reduce}_{C\to\gamma}\ \mathsf{g})\ \mathsf{ts} \mid C \to \gamma \in \mathbf{P} \,\} \\
 & & \\
\mathsf{reduce}_{C\to\gamma} & :: & (\bar{C} \to \mathsf{Parser\ r}) \to (\bar{\gamma} \to \mathsf{Parser\ r}) \\
\mathsf{reduce}_{C\to\gamma}\ \mathsf{g}\ v_\gamma & = & \mathsf{g}\ (\mathsf{Rule}_{C\to\gamma}\ v_\gamma) \\
 & & \\
\mathsf{goto}_{A\to\alpha\cdot X\beta} & :: & (\overline{X\beta} \to \mathsf{Parser\ r}) \to (\bar{X} \to \mathsf{Parser\ r}) \\
\mathsf{goto}_{A\to\alpha\cdot X\beta}\ \mathsf{k}\ v_X & = & \mathsf{state}_{A\to\alpha X\cdot\beta}\ (\mathsf{k}\ v_X)
\end{array}
$$

Figure 6. Equational characterization of state, shift and goto.

for the expression grammar is listed in Appendix A. It loops, however, in Haskell since there are two nested infinite branches (due to the left recursion in $E \to E{+}T$ and $T \to T{*}F$). A system guaranteeing fair selection of alternatives as in some logic programming systems would ensure that all solutions were produced, but would sometimes signal failure with nontermination.

5. Non-deterministic recursive ascent parsing

As noted above, the equations for items of the forms $A \to \alpha\cdot$ and $A \to \alpha \cdot c\beta$ are directly implementable, corresponding to the reduce and shift actions of a traditional LR parser. It remains to eliminate actions of the form $A \to \alpha \cdot C\beta$ by expanding them to reduce or shift actions. For this step, we operate on the continuation-passing functions $\mathtt{state}$ and $\mathtt{shift}_C$ expressed in a point-free style:

$$\begin{array}{lcl}
\mathtt{state}_{A\to\alpha\cdot} & = & \mathtt{id} \\
\mathtt{state}_{A\to\alpha\cdot X\beta} & = & \mathtt{shift}_X \circ \mathtt{goto}_{A\to\alpha\cdot X\beta} \\
\mathtt{shift}_C & = & \bigcup \{\, \mathtt{state}_{C\to\cdot\gamma} \circ \mathtt{reduce}_{C\to\gamma} \mid C \to \gamma \in \mathbf{P} \,\}
\end{array}$$

where in the last equation union is lifted pointwise to function spaces: $(f \cup g)\ x = f\ x \cup g\ x$. Each $\mathtt{state}$ and each $\mathtt{shift}_C$ expands to the union of a possibly infinite set of compositions of finite sequences of $\mathtt{goto}$, $\mathtt{shift}_c$ and $\mathtt{reduce}$ functions. We shall denote these sets of sequences by nonterminal symbols $\mathtt{State}$ and $\mathtt{Shift}_C$ respectively, defined by a left-linear grammar corresponding to the above equations:

$$\begin{array}{lcll}
\mathtt{State}_{A\to\alpha\cdot} & \to & \epsilon & \\
\mathtt{State}_{A\to\alpha\cdot X\beta} & \to & \mathtt{Shift}_X\ \mathtt{goto}_{A\to\alpha\cdot X\beta} & \\
\mathtt{Shift}_c & \to & \mathtt{shift}_c & \\
\mathtt{Shift}_C & \to & \mathtt{State}_{C\to\cdot\gamma}\ \mathtt{reduce}_{C\to\gamma} & C \to \gamma \in \mathbf{P}
\end{array}$$

From this perspective, the task of eliminating left recursion reduces to the standard procedure of transforming this left-linear grammar into an equivalent right-linear grammar (see Section 2).

There is, however, a caveat. The transformation only preserves the meaning of the start symbol. Therefore, we confine our attention to $\mathtt{State}_q$ for an arbitrary, but fixed kernel item q. The derivation below must then be repeated for every kernel item. (We need not consider non-kernel items though.) We furthermore restrict the grammar to symbols reachable from $\mathtt{State}_q$ as this does not change the language denoted by $\mathtt{State}_q$. Then $\mathtt{State}_q$ is given by the following productions (for readability we have added a new start symbol $\mathtt{Start}$):

$$\begin{array}{lcll}
\mathtt{Start} & \to & \mathtt{State}_q & \\
\mathtt{State}_i & \to & \epsilon & i = A \to \alpha\cdot \in q^* \\
\mathtt{State}_i & \to & \mathtt{Shift}_X\ \mathtt{goto}_i & i \in q^*/X \\
\mathtt{Shift}_c & \to & \mathtt{shift}_c & c \in \mathit{next}(q^*) \\
\mathtt{Shift}_C & \to & \mathtt{State}_{C\to\cdot\gamma}\ \mathtt{reduce}_{C\to\gamma} & C \to \cdot\gamma \in q^+
\end{array}$$

The equivalent right-linear grammar is then

$$\begin{array}{lll}
\overline{\mathsf{State}}_q & \to \epsilon & \\
\mathsf{Start} & \to \overline{\mathsf{State}}_i & i = A \to \alpha\cdot \in q^* \\
\overline{\mathsf{Shift}}_X & \to \mathsf{goto}_i\,\overline{\mathsf{State}}_i & i \in q^*/X \\
\mathsf{Start} & \to \mathsf{shift}_c\,\overline{\mathsf{Shift}}_c & c \in \mathit{next}(q^*) \\
\overline{\mathsf{State}}_{C\to\cdot\gamma} & \to \mathsf{reduce}_{C\to\gamma}\,\overline{\mathsf{Shift}}_C & C \to \cdot\gamma \in q^+
\end{array}$$

Reverting the right-linear grammar back into a program we obtain (state_q corresponds to Start):

$$\begin{array}{lll}
\mathsf{state}_q & = \bigcup\{\, \mathsf{shift}_c \circ \overline{\mathsf{shift}}_c \mid c \in \mathit{next}(q^*)\,\} \cup \\
& \quad \bigcup\{\, \overline{\mathsf{state}}_i \mid i = A \to \alpha\cdot \in q^*\,\} \\
\overline{\mathsf{state}}_q & = \mathsf{id} \\
\overline{\mathsf{state}}_{C\to\cdot\gamma\in q^+} & = \mathsf{reduce}_{C\to\gamma} \circ \overline{\mathsf{shift}}_C \\
\overline{\mathsf{shift}}_X & = \bigcup\{\, \mathsf{goto}_i \circ \overline{\mathsf{state}}_i \mid i \in q^*/X\,\}
\end{array}$$

Note that the transformation relies on the continuity of shift, reduce and goto. Furthermore, it is important to note that the two equations for $\overline{\mathsf{state}}$ do not overlap because $q \notin q^+$ for any kernel item q.

Now, for the following it is more convenient to rewrite the definition above into an applicative form.

$$\begin{array}{lll}
\mathsf{state}_q\ \mathsf{k} & = \bigcup\{\, \mathsf{shift}_c\ (\overline{\mathsf{shift}}_c\ \mathsf{k}) \mid c \in \mathit{next}(q^*)\,\} \cup \\
& \quad \bigcup\{\, \overline{\mathsf{state}}_i\ k \mid i = A \to \alpha\cdot \in q^*\,\} \\
\overline{\mathsf{state}}_q\ \mathsf{k} & = \mathsf{k} \\
\overline{\mathsf{state}}_{C\to\cdot\gamma\in q^+}\ \mathsf{k} & = \mathsf{reduce}_{C\to\gamma}\ (\overline{\mathsf{shift}}_C\ \mathsf{k}) \\
\overline{\mathsf{shift}}_X\ \mathsf{k} & = \bigcup\{\, \mathsf{goto}_i\ (\overline{\mathsf{state}}_i\ \mathsf{k}) \mid i \in q^*/X\,\}
\end{array}$$

Since we are aiming at a single definition for state q, we turn $\overline{\mathsf{state}}$ and $\overline{\mathsf{shift}}$ into local definitions setting $\mathsf{k}_i = \overline{\mathsf{state}}_i\ \mathsf{k}$ and $\mathsf{g}_X = \overline{\mathsf{shift}}_X\ \mathsf{k}$:

$$\begin{array}{lll}
\mathsf{state}_q\ \mathsf{k} & = \bigcup\{\, \mathsf{shift}_c\ \mathsf{g}_c \mid c \in \mathit{next}(q^*)\,\} \cup \\
& \quad \bigcup\{\, \mathsf{k}_i \mid i = A \to \alpha\cdot \in q^*\,\} \\
\quad \textbf{where}\ \mathsf{k}_q & = \mathsf{k} \\
\mathsf{k}_{C\to\cdot\gamma\in q^+} & = \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}_C \\
\mathsf{g}_X & = \bigcup\{\, \mathsf{goto}_i\ \mathsf{k}_i \mid i \in q^*/X\,\}
\end{array}$$

The last transformation is also known as λ-dropping (Dan99). We can finally expand g_X by calculating

$$\begin{array}{ll}
& \mathsf{g}_X\ \mathsf{v}_X \\
= & \quad \langle\ \text{definition of } \mathsf{g}_X\ \rangle \\
& \bigcup\{\, \mathsf{goto}_i\ \mathsf{k}_i\ \mathsf{v}_X \mid i = A \to \alpha\cdot X\beta \in q^*\,\} \\
= & \quad \langle\ \text{definition of } \mathsf{goto}_i\ \rangle \\
& \bigcup\{\, \mathsf{state}_{A\to\alpha X\cdot\beta}\ (\mathsf{k}_i\ \mathsf{v}_X) \mid i = A \to \alpha\cdot X\beta \in q^*\,\} \\
= & \quad \langle\ \text{definition of } Q{+}X \text{ and } Q/X\ \rangle \\
& \bigcup\{\, \mathsf{state}_{i+X}\ (\mathsf{k}_i\ \mathsf{v}_X) \mid i \in q^*/X\,\}
\end{array}$$

$$
\begin{array}{lcl}
\mathsf{state}_{A\to\alpha\cdot\beta} :: (\bar{\beta} \to \mathsf{Parser\ r}) \to \mathsf{Parser\ r} & & \\
\mathsf{state}_q\ \mathsf{k}_q\ \mathsf{ts} & = & \bigcup\{\, \mathsf{shift}_c\ \mathsf{g}_c\ \mathsf{ts} \mid c \in \mathit{next}(q^*)\,\} \cup \\
& & \bigcup\{\, \mathsf{k}_i\ \mathsf{ts} \mid i = A \to \alpha\cdot \in q^*\,\} \\
\quad \textbf{where}\ \mathsf{k}_{C\to\cdot\gamma\in q^+} & = & \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}_C \\
\quad\ \ \mathsf{g}_X\ \mathsf{v}_X & = & \bigcup\{\, \mathsf{state}_{i+X}\ (\mathsf{k}_i\ \mathsf{v}_X) \mid i \in q^*/X\,\}
\end{array}
$$

Figure 7. Non-deterministic LR(0) parser for a kernel item q.

$$
\begin{array}{lcl}
\mathsf{state}_Q :: (\bar{\beta} \to \mathsf{Parser\ r} \mid A \to \alpha\cdot\beta \in Q) \to \mathsf{Parser\ r} & & \\
\mathsf{state}_Q\ (\,\mathsf{k}_q \mid q \in Q\,)\ \mathsf{ts} & = & \bigcup\{\, \mathsf{shift}_c\ \mathsf{g}_c\ \mathsf{ts} \mid c \in \mathit{next}(Q^*)\,\} \cup \\
& & \bigcup\{\, \mathsf{k}_i\ \mathsf{ts} \mid i = A \to \alpha\cdot \in Q^*\,\} \\
\quad \textbf{where}\ \mathsf{k}_{C\to\cdot\gamma\in Q^+} & = & \mathsf{reduce}_{C\to\gamma}\ \mathsf{g}_C \\
\quad\ \ \mathsf{g}_X\ \mathsf{v}_X & = & \mathsf{state}_{Q^*+X}\ (\,\mathsf{k}_i\ \mathsf{v}_X \mid i \in Q^*/X\,)
\end{array}
$$

Figure 8. Non-deterministic LR(0) parser for a set of kernel items Q.

We have derived the parser of Figure 7. The parser for our running example, the expression grammar, can be found in Appendix B. Although our parsers are not yet deterministic, a relationship to traditional table-driven parsers can be noted: the right-hand side of state_q corresponds to an entry in the action table, the local function g_X to an entry of the goto table.

6. Deterministic recursive ascent parsing

6.1. Pursuing sets of items in parallel

The first step towards a deterministic parser is to collect sets of parsers for individual kernel items into a parser for a set Q of kernel items, with a corresponding tuple of parsing continuations:

$$\mathsf{state}_Q\ (\,\mathsf{k}_q \mid q \in Q\,)\ \mathsf{ts} \;=\; \bigcup\{\, \mathsf{state}_q\ \mathsf{k}_q\ \mathsf{ts} \mid q \in Q\,\}$$

This corresponds to the classical subset construction for converting a non-deterministic finite automaton to a deterministic one. Since everything distributes over union, we obtain the parser of Figure 8.

Using the properties of **case**, the unions of shift_c functions may furthermore be combined into a single **case** statement:

$$
\begin{array}{lcl}
\mathsf{shift}_{c_1}\ \mathsf{g}_1\ \mathsf{ts} \cup \cdots \cup \mathsf{shift}_{c_n}\ \mathsf{g}_n\ \mathsf{ts} \;=\; \textbf{case}\ \mathsf{ts}\ \textbf{of} & & \\
\qquad \mathsf{T}_{c_1}\ \mathsf{v}_{c_1} : \mathsf{tr} & \to & \mathsf{g}_1\ \mathsf{v}_{c_1}\ \mathsf{tr} \\
\qquad \cdots & & \\
\qquad \mathsf{T}_{c_n}\ \mathsf{v}_{c_n} : \mathsf{tr} & \to & \mathsf{g}_n\ \mathsf{v}_{c_n}\ \mathsf{tr} \\
\qquad \_ & \to & \varnothing
\end{array}
$$

The state functions in our parsers take parsing continuations as arguments, one for each kernel item in the corresponding state of the LR automaton. Traditional parsers also maintain a stack of attribute values, popping the required number of values when a reduction occurs. In contrast, in our parsers the parsing continuations are applied to these attributes as they become available. In effect, we have a short value stack for each continuation. This seems expensive, but the number of continuations of a state is equal to the number of kernel items, and in typical parsers this is small.

6.2. Using look-ahead

The remaining non-determinism of the parser in Figure 8 arises when a state contains both shifts and reductions (shift-reduce conflict) or more than one reduction (reduce-reduce conflict). By definition, these conflicts do not occur if the grammar is LR(0). For these grammars we have constructed a deterministic parser: the set of results is either empty or has a single element. We can represent such sets using Haskell's `Maybe` type. For non-LR(0) grammars we can use look-ahead information to resolve these conflicts. For example, in the expression parser the item set $\{ E \to E\texttt{+}T \cdot, T \to T \cdot \texttt{*}F \}$ yields the state function

```
state6 :: Parser r → (MulOp → Factor → Parser r) → Parser r
state6 k_{E→E+T·} k_{T→T·*F} ts =
  (case ts of MULOP v : tr → state8 (k_{T→T·*F} v) tr
              _            → ∅)
  ∪ k_{E→E+T·} ts   -- shift-reduce conflict
```

The look-ahead set for the reduce item $\mathsf{k}_{E \to E\texttt{+}T \cdot}$ is {RPAREN, ADDOP, EOF}. Using this information we can refine the case expression to

```
state6 :: Parser r → (MulOp → Factor → Parser r) → Parser r
state6 k_{E→E+T·} k_{T→T·*F} ts =
  case ts of MULOP v : tr  → state8 (k_{T→T·*F} v) tr
             ADDOP v : _   → k_{E→E+T·} ts
             RPAREN  : _   → k_{E→E+T·} ts
             EOF     : _   → k_{E→E+T·} ts
             _             → ∅
```

Eliminating reductions in this way removes many conflicts. If the grammar is LALR(1), it makes the parser deterministic. Alternatively, we can perform the reduction on all symbols other than MULOP.

```
state6 :: Parser r → (MulOp → Factor → Parser r) → Parser r
state6 k_{E→E+T·} k_{T→T·*F} ts =
  case ts of MULOP v : tr  → state8 (k_{T→T·*F} v) tr
             _             → k_{E→E+T·} ts
```

This choice corresponds to a standard optimization of table-based LR parsers (ASU86), where the most common reduce action is generalized. The resulting expression parser is given in Appendix C.

6.3. Smaller parsers

For realistic grammars, the parsers generated by the above method are fast, but quite large, possibly stretching the compiler or the available memory. However, several simple steps can drastically reduce the size of the parser. Since our parsers eschew tables, exposing the control flow, a reasonable optimizing compiler would perform many of these steps anyway. However, our concern is that the input program would be too large for the compiler, so they should be done by the parser generator.

Firstly, because in our parsers the parsing continuations are already applied to past values, the only part of a kernel item $A \to \alpha \cdot \beta$ that matters is β. If these parts of the items, and the associated look-aheads, are identical, the states can be merged. For example, in the parser of Appendix C, states 6 and 7 are identical. The whole items, however, do not match, so the states could not be merged in a table-based parser.

The extreme case, where a state consists of a single item, and that of the form $A \to \alpha \cdot$, is quite common. In this case, the state is an identity function, and thus may be eliminated. In the example parser, states 3, 9, 10, 12 and 13 may be eliminated in this way.

Now note that states 1, 5, 8 and 11 have identical actions, differing only in the goto functions they use. So one could abstract out the common action part, and parameterize it by the goto functions and continuations used. This is a common situation, and corresponds to a standard optimization for table-based parsers (ASU86). In this example, this common part is equivalent to state_8 (after the previous optimization):

$$
\begin{array}{l}
\mathsf{state}_8 :: (\mathsf{Factor} \to \mathsf{Parser}\ \mathsf{r}) \to \mathsf{Parser}\ \mathsf{r} \\
\mathsf{state}_8\ \mathsf{k}_{T \to T * \cdot F}\ \mathsf{ts} \qquad = \\
\quad \textbf{case}\ \mathsf{ts}\ \textbf{of}\ \mathsf{LPAREN} : \mathsf{tr} \to \mathsf{state}_{11}\ (\mathsf{reduce}_{F \to (\,E\,)}\ \mathsf{g}_F)\ \mathsf{tr} \\
\qquad\qquad\quad \mathsf{IDENT}\ \mathsf{v} : \mathsf{tr} \to \mathsf{reduce}_{F \to \mathtt{id}}\ \mathsf{k}_{T \to T * \cdot F}\ \mathsf{v}\ \mathsf{tr} \\
\qquad\qquad\quad \_ \qquad\qquad\ \to \varnothing
\end{array}
$$

We can go further: the definition of g_T and g_F in states 1 and 11 is the same as in state 5, so they may be defined in terms of it. (This could also be done in a table-based parser, but would involve slowing down the selection of the goto function.) The resulting expression parser is given in Appendix D.

```
type Environment = [(String, Int)]
type Expr'       = Environment → Int
type Term'       = Environment → Int
type Factor'     = Environment → Int
rule_{E→E+T}                    :: Expr' → AddOp → Term' → Expr'
rule_{E→E+T} e Plus t env       = e env + t env
rule_{E→E+T} e Minus t env      = e env − t env
rule_{E→T}                      :: Term' → Expr'
rule_{E→T}                      = id
rule_{T→T*F}                    :: Term' → MulOp → Factor' → Term'
rule_{T→T*F} t Times f env      = t env * f env
rule_{T→T*F} t Divide f env     = t env `div` f env
rule_{T→F}                      :: Factor' → Term'
rule_{T→F}                      = id
rule_{F→(E)}                    :: Expr' → Factor'
rule_{F→(E)}                    = id
rule_{F→id}                     :: String → Factor'
rule_{F→id} n env               = fromJust (lookup n env)
```

Figure 9. Attribute functions for an environment-based evaluator.

6.4. Attributes

In many applications, we will want to apply a fold to the parse tree returned by our parser. A simple form of deforestation (GLP83) is to redefine the attribute type $\bar{A}$ of each nonterminal A and replace each $\mathsf{Rule}_{A\to\alpha}$ constructor with a corresponding function $\mathsf{rule}_{A\to\alpha}$. Since the `Rule` constructors occur only in the `reduce` functions, only these need be changed, in addition to redefining the types.

For example, if we wished to evaluate a parsed expression in an environment mapping identifiers to integer values, we could use the definitions of Figure 9.

7. Related work on functional LR parsers

Leermakers, Augusteijn and Kruseman Aretz (LAK92; Lee93; Aug93) defined several functional parsers, including an LR parser consisting of functions $[Q]$ for a set Q of kernel items specified as

$$[Q]\ w = \{\ (A \to \alpha \cdot \beta, w_2) \mid A \to \alpha \cdot \beta \in Q \wedge \beta \to^* w_1 \wedge w = w_1 w_2\ \}$$

Many nondeterministic recursive functions, including the functions $[Q]$ above, are amenable to a group of general transformations studied

$$
\begin{array}{lcl}
[Q] & :: & [\mathsf{Token}] \to \mathbb{P}\,(\mathsf{Item}, [\mathsf{Token}]) \\
[Q]\ w & = & \overline{[Q]}_c\ w' \cup \\
 & & \{\,(A \to \alpha\cdot, w) \mid A \to \alpha\cdot \in Q\,\} \cup \\
 & & \bigcup \left\{\, \overline{[Q]}_C\ w \mid C \to \cdot \in Q^+ \,\right\} \\
 & & \textbf{where}\ (c : w') = w \\
\overline{[Q]}_X & :: & [\mathsf{Token}] \to \mathbb{P}\,(\mathsf{Item}, [\mathsf{Token}]) \\
\overline{[Q]}_X\ w & = & \{\,(i, w') \mid i \in Q/X \wedge (i{+}X, w') \in [Q^*{+}X]\ w\,\} \cup \\
 & & \bigcup \left\{\, \overline{[Q]}_C\ w' \mid j \in Q^+/X \wedge (j{+}X, w') \in [Q^*{+}X]\ w \,\right\}
\end{array}
$$

Figure 10. Direct-style functional parser of Leermakers.

by Augusteijn (Aug93). These transformations lead to the definition of Figure 10, where we have omitted handling of attributes to avoid clutter. Note a minor complication in the definition of $[Q]$, where kernel reduce items $A \to \alpha\cdot$ and predict reduce items $C \to \cdot$ are treated differently.

This version was further refined to an imperative implementation, providing a derivation of the parsers of Kruseman Aretz (Kru88) and Pennello (Pen86).

Leermakers also defined a continuation-passing version, where the continuation is passed the pair $(A \to \alpha\cdot\beta, w_2)$, and then winds back through $|\alpha|$ continuations and dispatches on the nonterminal A, corresponding to the traditional reduction and goto function.

Sperber and Thiemann (ST00) also derived a continuation-passing version, by applying partial evaluation techniques to the parser of Figure 10. Their variant passes a list of continuations, corresponding to the stack of states (goto functions) in traditional parsers. They realized that only a finite number of these were needed (the largest $|\alpha|$ for any item $A \to \alpha\cdot\beta$ in Q) so they could be passed as separate arguments. A consequence of these transformations is that kernel and predict items may be handled uniformly, as in our parser. They handled attributes using a stack of values, which may be similarly truncated. Sperber and Thiemann worked in an untyped setting, but their parser may be recast in typed form as in Figure 11. Each k_i here corresponds to a row of the goto table of a traditional parser, which is usually quite sparse. On reduction, the parser selects the continuation, which then dispatches over the nonterminal. In comparison, our parser passes entries of the goto table as separate arguments.

It is possible to remove the interpretive overhead from Sperber and Thiemann's parser by fusing $\mathsf{k}_{|\gamma|} \circ \mathsf{N}_C$. In this way, we obtain the parser

$$\begin{array}{l}
\textbf{data}\ \mathsf{NonTerminal} \;=\; \sum_{A \in \mathbf{N}^\dagger} \mathsf{N}_A\ \bar{A} \\
\\
[Q] \;::\; (\mathsf{NonTerminal} \to \mathsf{Parser\ r})^{|\alpha|} \to \bar{\alpha} \to \mathsf{Parser\ r} \\
[Q]\ \mathsf{k}_1\ \ldots\ \mathsf{k}_{|\alpha|}\ v_\alpha\ \mathsf{ts} \;=\; \bigcup \{\, \mathsf{shift}_c\ \mathsf{g}_c\ \mathsf{ts} \mid c \in \mathit{next}(Q^*) \,\} \\
\qquad\qquad\qquad\qquad \cup\ \bigcup \{\, \mathsf{reduce}_{C \to \gamma}\ (\mathsf{k}_{|\gamma|} \circ \mathsf{N}_C)\ v_\gamma\ \mathsf{ts} \mid C \to \gamma\cdot \in Q^* \,\} \\
\quad \textbf{where}\ \mathsf{g}_X\ v_X \;=\; [Q^*{+}X]\ \mathsf{k}_0\ \mathsf{k}_1\ \ldots\ \mathsf{k}_{|\alpha'|}\ v_{\alpha'}\ v_X \\
\qquad\quad \mathsf{k}_0\ (\mathsf{N}_A\ v_A) \;=\; \mathsf{g}_A\ v_A \qquad\qquad A \in \mathit{next}(Q^*) \\
\qquad\quad \alpha \text{ maximal such that } A \to \alpha \cdot \beta \in Q \\
\qquad\quad \alpha' \text{ maximal such that } A' \to \alpha' \cdot X\beta' \in Q^*
\end{array}$$

Figure 11. Typed version of Sperber and Thiemann's parser for a set of kernel items Q.

$$\begin{array}{l}
[Q] \;::\; (\bar{A} \to \mathsf{Parser\ r} \mid A \to \alpha \cdot \beta \in Q) \to \bar{\alpha}_{\max} \to \mathsf{Parser\ r} \\
[Q]\ (\mathsf{k}_i \mid i \in Q)\ v_{\alpha_{\max}}\ \mathsf{ts} \;=\; \bigcup \{\, \mathsf{shift}_c\ \mathsf{g}_c\ \mathsf{ts} \mid c \in \mathit{next}(Q^*) \,\} \\
\qquad\qquad\qquad\qquad\quad \cup\ \bigcup \{\, \mathsf{reduce}_{C \to \gamma}\ \mathsf{k}_{C \to \gamma\cdot}\ v_\gamma\ \mathsf{ts} \mid C \to \gamma\cdot \in Q^* \,\} \\
\quad \textbf{where}\ \mathsf{g}_X\ v_X \;=\; [Q^*{+}X]\ (\mathsf{k}_i \mid i \in Q^*/X)\ v_{\alpha'_{\max}}\ v_X \\
\qquad\quad \mathsf{k}_{C \to \cdot\gamma} \;=\; \mathsf{g}_C \\
\qquad\quad \alpha_{\max} \text{ maximal such that } A \to \alpha \cdot \beta \in Q \\
\qquad\quad \alpha'_{\max} \text{ maximal such that } A' \to \alpha' \cdot X\beta' \in Q^*
\end{array}$$

Figure 12. Intermediate parser for a set of kernel items Q.

of Figure 12. From this, we could then obtain the parser of Figure 8 by applying to continuations to the values as they become available, instead of passing the top portion of the stack.

Another scheme for typing LR parsers was presented by Pottier and Régis-Gianas (PRG05). They also address the interpretive overhead of the union type for nonterminals, but their solution requires a type system extended with inductive type families (also known as generalized algebraic data types).

A different approach is pursued by Pepper (Pep04). Though recursive descent formulations of grammars are not always directly executable, Pepper shows that if the underlying language is LR(k), the programs may be transformed into equivalent recursive descent parsers, extended with a delayed choice operator.

8. Measurements

The purpose of this section is to compare the time and space behaviour of our parsers with that of traditional table-based ones. To get

grammar[a]	parser	src[b]	bin[c]	prg[d]	time[e]	%[f]	%[g]
expr 1.0M 0.100s	`frown -cs`	5.0K	18K	482K	2.184s	101	101
	`frown`	5.7K	18K	482K	2.160s	100	100
	`frown -cc`	4.3K	20K	484K	2.264s	105	105
	`frown -cg`	5.5K	25K	491K	2.480s	115	116
	`happy`	12K	27K	492K	6.784s	314	324
	`happy -cg`	14K	19K	482K	4.380s	203	208
	`happy -acg`	14K	16K	488K	4.772s	221	227
Oberon 2.9M 1.492s	`frown -cs`	114K	177K	1.2M	2.568s	100	100
	`frown`	169K	341K	1.4M	2.616s	102	104
	`frown -cc`	77K	242K	1.3M	2.620s	102	105
	`frown -cg`	118K	237K	1.3M	2.636s	103	106
	`happy`	87K	215K	1.2M	3.700s	144	205
	`happy -cg`	109K	122K	1.1M	4.180s	163	250
	`happy -acg`	84K	122K	1.1M	3.736s	145	209
Haskell 1.9M 3.016s	`frown -cs`	414K	1.2M	2.6M	4.172s	100	100
	`frown`	732K	2.1M	3.4M	4.272s	102	109
	`frown -cc`	298K	1.2M	2.6M	4.256s	102	107
	`frown -cg`	307K	1.0M	2.3M	4.204s	101	103
	`happy`	207K	666K	2.0M	5.068s	121	178
	`happy -cg`	243K	329K	1.7M	5.836s	140	244
	`happy -acg`	163K	205K	1.5M	5.512s	132	216

[a]grammar, size of the input to the parser, lexing time (minimum of 4 runs)
[b]size of the generated parser (source) [c]size of the generated parser (binary)
[d]size of the executable (including lexer and pretty printer)
[e]execution time (minimum of 4 runs, including lexing and pretty printing)
[f]time relative to the fastest parser (including lexing and pretty printing)
[g]time relative to the fastest parser (excluding lexing)

Figure 13. Benchmark results.

a somewhat broader picture we also include alternative non-table based parsers in the competition. Table 13 lists the results of the benchmarks, which were conducted on an AMD Athlon 64 3000+ with 2GB of main memory. We consider three grammars, a tiny, a medium-sized and a large grammar:

1. the expression grammar of Example 1 (7 productions, 13 states),
2. a grammar for Oberon (175 productions, 288 states),
3. a grammar for Haskell 98 (277 productions, 482 states).

The parsers for these grammars have been generated using two parser generators for Haskell: `happy` (GM05) is a yacc-like parser generator, `frown` (Hin05) is an LALR(k) parser generator developed by the first

author of the present paper. The generated parsers were compiled using the Glasgow Haskell Compiler, GHC 6.4.1 (The05), with optimizations, `-O`, turned on. Here are the salient features of the generated parsers:

1. `frown -cs` : the parsers developed in this paper;
2. `frown` : a table-free parser that uses an explicit stack of *state transitions* (represented by a tailor-made union type (DS00));
3. `frown -cc` : like 2 but with a more conventional stack type (a tailor-made union type for an alternation of states and attributes);
4. `frown -cg` : like 2 with the stack represented by a nested pair type (similar to the version of Figure 11 but passing the whole stack rather than a window of the top portion);
5. `happy` : a table-free parser that uses a polymorphic stack and a universal type for attributes (the union of all attribute types);
6. `happy -cg` : like 5, but using several GHC extensions including unboxed integers and unsafe type coercions (to avoid the need for a union type);
7. `happy -acg`: a table-based parser that uses the same GHC extensions as 6.

The different parsers for a grammar share the same lexical analyzer; the parsing actions construct the abstract syntax tree of the input (not the parse tree), which is then pretty-printed.

For tiny and medium-sized grammars the parsers derived in this paper are the implementation of choice: they are fast, at least twice as fast as table-based parsers, and small, the size commensurate to that of table-based ones. For large grammars, the situation is not as clear cut: our parsers are still the fastest; the size of the generated code, however, exceeds that of table-based parsers: the executable is roughly 70% larger. In a sense, the findings mirror the typical time/space trade-off between interpreted and compiled code. It should be noted, however, that the `frown` implementation does not yet incorporate the optimizations described in Section 6.3. The merging of states might ameliorate the size problem.

To summarize, the stackless, table-free parsers are consistently twice as fast as table-based parsers; the larger the grammar the larger the size of the generated parser relative to the table-based one.

As a final remark, it is interesting to note that the use of unsafe features does not necessarily improve the running time. Or to put it

positively, it is pleasing to see that a clean derivation gives rise to an efficient parser.

9. Conclusion

We have derived a purely functional implementation of LR parsing. The derived parsers consist of mutually recursive functions corresponding to the states of the underlying LR automaton. Each function takes a number of continuation arguments corresponding to the kernel items of that state. No separate stack is needed as the parsing continuations are immediately applied to the attributes when they become available. Furthermore, the parsers can be typed in a standard Hindley-Milner type system. Polymorphism is only required if one wants to support parsers with multiple entry points.

Once the stage has been set, the derivation proceeds surprisingly smoothly with little or no Eureka steps: the first major transformation (Section 5) is motivated by the need to eliminate the left-recursion present in the initial LL parser; the second major transformation (Section 6.1) eliminates the non-determinism of the intermediate parser, provided the grammar is LR(0).

An interesting feature of our parsers is the passing multiple continuations. These could also be implemented using the multi-function return feature proposed by Shivers (SFar) as an alternative to continuations.

Initial measurements are encouraging: compared to conventional table-based parsers, our parsers are roughly twice as fast. This is not astonishing as our implementation avoids the interpretative overhead of the traditional approach. Furthermore, since our parsers expose the control flow, they are amenable to several optimizations, some of which are specific to the functional representation. On the negative side, the generated parsers are fairly large for realistic grammars. We are confident, however, that the optimizations described in Section 6.3 will ameliorate this problem.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume I: Parsing. Prentice-Hall, 1972.

Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, TU Eindhoven, September 1993.

Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Aart Middeldorp and Taisuke Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, volume 1722, pages 241–250, November 1999.

Luc Duponcheel and Doaitse Swierstra. A functional program for generating efficient functional LALR(1) parsers, September 2000. unpublished note.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Proc. Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, 1983. ACM.

Andy Gill and Simon Marlow. *Happy: The Parser Generator for Haskell, Version 1.15*, 2005. Available from `http://haskell.org/happy/`.

Ralf Hinze. Constructing tournament representations: An exercise in pointwise relational programming. In Eerke A. Boiten and Bernhard Möller, editors, *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002), Dagstuhl, Germany, July 8-10, 2002*, volume 2386, pages 131–147, July 2002.

Ralf Hinze. *Frown — an LALR(k) parser generator for Haskell, Version 0.6*, 2005. Available from `http://www.informatik.uni-bonn.de/~ralf/frown/`.

Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

F.E.J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29:201–206, 1988.

René Leermakers, Lex Augusteijn, and Frans E.J. Kruseman Aretz. A functional LR-parser. *Theoretical Computer Science*, 104:313–323, 1992.

René Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, July 1993.

Thomas J. Pennello. Very fast LR parsing. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 145–151, New York, NY, USA, 1986. ACM Press.

Peter Pepper. How to obtain powerful parsers that are elegant and practical, March 2004.

Simon Peyton Jones. *Haskell 98 Language and Libraries*. 2003.

Fran cois Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. In *ACM SIGPLAN Workshop on ML*, Electronic Notes in Theoretical Computer Science, pages 149–173, September 2005.

Olin Shivers and David Fisher. Multi-return function call. *Journal of Functional Programming*, to appear.

Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(3):224–264, 2000.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4.1*, 2005. Available from `http://www.haskell.org/ghc/`.

P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Verlag, 1988.

Appendix

In the appendix we shall abbreviate Parser by $\mathscr{P}$.

A. LL(0) Expression Parser

The function parse_E is a non-deterministic recursive descent parser for the expression grammar (Example 1). The parser loops, however, for non-trivial inputs as the functions $\mathsf{state}_{E\to\,.\,E+T}$ and $\mathsf{state}_{T\to\,.\,T*F}$ are left-recursive.

```
parse_E              :: P Expr
parse_E              = state_{E†→ . E$} accept
state_{E†→ . E$}     :: (Expr → P r) → P r
state_{E†→ . E$} k ts = state_{E→ . E+T} (reduce_{E→E+T} g_E) ts
                     ∪ state_{E→ . T} (reduce_{E→T} g_E) ts
  where g_E v        = state_{E†→E . $} (k v)
state_{E†→E . $}     :: P r → P r
state_{E†→E . $} k ts = case ts of EOF : tr →   k tr
                                   _        →   ∅
state_{E→ . E+T}     :: (Expr → AddOp → Term → P r) → P r
state_{E→ . E+T} k ts = state_{E→ . E+T} (reduce_{E→E+T} g_E) ts   -- left recursion
                     ∪ state_{E→ . T} (reduce_{E→T} g_E) ts
  where g_E v        = state_{E→E . +T} (k v)
state_{E→E . +T}     :: (AddOp → Term → P r) → P r
state_{E→E . +T} k ts = case ts of ADDOP v : tr → state_{E→E+ . T} (k v) tr
                                   _            → ∅
state_{E→E+ . T}     :: (Term → P r) → P r
state_{E→E+ . T} k ts = state_{T→ . T*F} (reduce_{T→T*F} g_T) ts
                     ∪ state_{T→ . F} (reduce_{T→F} g_T) ts
  where g_T v        = state_{E→E+T . } (k v)
state_{E→E+T . }     :: P r → P r
state_{E→E+T . } k ts = k ts
state_{E→ . T}       :: (Term → P r) → P r
state_{E→ . T} k ts  = state_{T→ . T*F} (reduce_{T→T*F} g_T) ts
                     ∪ state_{T→ . F} (reduce_{T→F} g_T) ts
  where g_T v        = state_{E→T . } (k v)
state_{E→T . }       :: P r → P r
state_{E→T . } k ts  = k ts
state_{T→ . T*F}     :: (Term → MulOp → Factor → P r) → P r
state_{T→ . T*F} k ts = state_{T→ . T*F} (reduce_{T→T*F} g_T) ts   -- left recursion
                     ∪ state_{T→ . F} (reduce_{T→F} g_T) ts
  where g_T v        = state_{T→T . *F} (k v)
```

```
state_{T→T·*F}        :: (MulOp → Factor → 𝒫 r) → 𝒫 r
state_{T→T·*F} k ts   = case ts of MULOP v : tr → state_{T→T*·F} (k v) tr
                                _               → ∅
state_{T→T*·F}        :: (Factor → 𝒫 r) → 𝒫 r
state_{T→T*·F} k ts   = state_{F→·(E)} (reduce_{F→(E)} g_F) ts
                      ∪ state_{F→·id} (reduce_{F→id} g_F) ts
  where g_F v         = state_{T→T*F·} (k v)
state_{T→T*F·}        :: 𝒫 r → 𝒫 r
state_{T→T*F·} k ts   = k ts
state_{T→·F}          :: (Factor → 𝒫 r) → 𝒫 r
state_{T→·F} k ts     = state_{F→·(E)} (reduce_{F→(E)} g_F) ts
                      ∪ state_{F→·id} (reduce_{F→id} g_F) ts
  where g_F v         = state_{T→F·} (k v)
state_{T→F·}          :: 𝒫 r → 𝒫 r
state_{T→F·} k ts     = k ts
state_{F→·(E)}        :: (Expr → 𝒫 r) → 𝒫 r
state_{F→·(E)} k ts   = case ts of LPAREN : tr → state_{F→(·E)} k tr
                                _              → ∅
state_{F→(·E)}        :: (Expr → 𝒫 r) → 𝒫 r
state_{F→(·E)} k ts   = state_{E→·E+T} (reduce_{E→E+T} g_E) ts
                      ∪ state_{E→·T} (reduce_{E→T} g_E) ts
  where g_E v         = state_{F→(E·)} (k v)
state_{F→(E·)}        :: 𝒫 r → 𝒫 r
state_{F→(E·)} k ts   = case ts of RPAREN : tr → state_{F→(E)·} k tr
                                _              → ∅
state_{F→(E)·}        :: 𝒫 r → 𝒫 r
state_{F→(E)·} k ts   = k ts
state_{F→·id}         :: (String → 𝒫 r) → 𝒫 r
state_{F→·id} k ts    = case ts of IDENT v : tr → state_{F→id·} (k v) tr
                                _              → ∅
state_{F→id·}         :: 𝒫 r → 𝒫 r
state_{F→id·} k ts    = k ts
```

Note that states with an identical string to the right of the dot are equal: $state_{E\to E+\cdot T} = state_{E\to\cdot T}$ and $state_{T\to T*\cdot F} = state_{T\to\cdot F}$.

B. LR(0) Expression Parser

The function $parse_E$ is a non-deterministic recursive ascent parser for the expression grammar. The non-determinism is exhibited by the locally defined g_X functions.

```
parse_E :: 𝒫 Expr
parse_E = state_{E†→·E$} accept
```

```
state_{E†→ · E$}                          :: (Expr → 𝒫 r) → 𝒫 r
state_{E†→ · E$} k_{E†→ · E$} ts           =
  case ts of LPAREN   : tr → state_{F→( · E)} (reduce_{F→( E)} g_F) tr
             IDENT ν  : tr → state_{F→id ·} (reduce_{F→id} g_F ν) tr
             _              → ∅
  where g_E ν                              = state_{E†→E · $} (k_{E†→ · E$} ν)
                                           ∪ state_{E→E · +T} (reduce_{E→E+T} g_E ν)
        g_T ν                              = state_{E→T ·} (reduce_{E→T} g_E ν)
                                           ∪ state_{T→T · *F} (reduce_{T→T*F} g_T ν)
        g_F ν                              = state_{T→F ·} (reduce_{T→F} g_T ν)
state_{E†→E · $} k ts                      =
  case ts of EOF      : tr → k tr
             _              → ∅
state_{E→E · +T} k ts                      =
  case ts of ADDOP ν : tr → state_{E→E+ · T} (k ν) tr
    _                    → ∅
state_{E→E+ · T}                           :: (Term → 𝒫 r) → 𝒫 r
state_{E→E+ · T} k_{E→E+ · T} ts           =
  case ts of LPAREN   : tr → state_{F→( · E)} (reduce_{F→( E)} g_F) tr
             IDENT ν  : tr → state_{F→id ·} (reduce_{F→id} g_F ν) tr
             _              → ∅
  where g_T ν                              = state_{E→E+T ·} (k_{E→E+ · T} ν)
                                           ∪ state_{T→T · *F} (reduce_{T→T*F} g_T ν)
        g_F ν                              = state_{T→F ·} (reduce_{T→F} g_T ν)
state_{E→E+T ·} k ts                       = k ts
state_{E→T ·} k ts                         = k ts
state_{T→T · *F} k ts                      =
  case ts of MULOP ν : tr → state_{T→T* · F} (k ν) tr
    _                    → ∅
state_{T→T* · F}                           :: (Factor → 𝒫 r) → 𝒫 r
state_{T→T* · F} k_{T→T* · F} ts           =
  case ts of LPAREN   : tr → state_{F→( · E)} (reduce_{F→( E)} g_F) tr
             IDENT ν  : tr → state_{F→id ·} (reduce_{F→id} g_F ν) tr
             _              → ∅
  where g_F ν                              = state_{T→T* F ·} (k_{T→T* · F} ν)
state_{T→T*F ·} k ts                       = k ts
state_{T→F ·} k ts                         = k ts
state_{F→ · ( E)} k ts                     =
  case ts of LPAREN   : tr → state_{F→( · E)} k tr
             _              → ∅
```

```
state_{F→(·E)}                    :: (Expr → 𝒫 r) → 𝒫 r
state_{F→(·E)} k_{F→(·E)} ts      =
  case ts of LPAREN : tr → state_{F→(·E)} (reduce_{F→(E)} g_F) tr
             IDENT ν : tr → state_{F→id·} (reduce_{F→id} g_F ν) tr
             _            → ∅
  where g_E ν               = state_{E→E·+T} (reduce_{E→E+T} g_E ν)
                            ∪ state_{F→(E·)} (k_{F→(·E)} ν)
        g_T ν               = state_{E→T·} (reduce_{E→T} g_E ν)
                            ∪ state_{T→T·*F} (reduce_{T→T*F} g_T ν)
        g_F ν               = state_{T→F·} (reduce_{T→F} g_T ν)
state_{F→(E·)} k ts               =
  case ts of RPAREN : tr → state_{F→(E)·} k tr
    _                  → ∅
state_{F→(E)·} k ts               = k ts
state_{F→·id} k ts                =
  case ts of IDENT ν : tr → state_{F→id·} (k ν) tr
             _            → ∅
state_{F→id·} k ts                = k ts
```

C. LALR(1) Expression Parser

The function $parse_E$ is a deterministic recursive ascent parser for the expression grammar corresponding to a conventional LALR(1) parser.

```
parse_E                           :: 𝒫 Expr
parse_E                           = state_1 accept
state_1                           :: (Expr → 𝒫 r) → 𝒫 r
state_1 k_{E†→·E$} ts             =
  case ts of LPAREN  : tr → state_11 (reduce_{F→(E)} g_F) tr
             IDENT ν : tr → state_13 (reduce_{F→id} g_F ν) tr
             _            → ∅
  where g_E ν               = state_2 (k_{E†→·E$} ν) (reduce_{E→E+T} g_E ν)
        g_T ν               = state_7 (reduce_{E→T} g_E ν) (reduce_{T→T*F} g_T ν)
        g_F ν               = state_10 (reduce_{T→F} g_T ν)
state_2                           :: 𝒫 r → (AddOp → Term → 𝒫 r) → 𝒫 r
state_2 k_{E†→E·$} k_{E→E·+T} ts  =
  case ts of ADDOP ν : tr → state_5 (k_{E→E·+T} ν) tr
             EOF     : tr → state_3 k_{E†→E·$} tr
             _            → ∅
state_3                           :: 𝒫 r → 𝒫 r
state_3 k_{E†→E$·} ts             = k_{E†→E$·} ts
```

```
state4                                  :: (AddOp → Term → 𝒫 r) → 𝒫 r → 𝒫 r
state4 k_{E→E·+T} k_{F→(E·)} ts         =
  case ts of RPAREN   : tr              → state12 k_{F→(E·)} tr
             ADDOP ν : tr               → state5 (k_{E→E·+T} ν) tr
             _                          → ∅
state5                                  :: (Term → 𝒫 r) → 𝒫 r
state5 k_{E→E+·T} ts                    =
  case ts of LPAREN   : tr              → state11 (reduce_{F→(E)} g_F) tr
             IDENT ν  : tr              → state13 (reduce_{F→id} g_F ν) tr
             _                          → ∅
  where g_T ν                           = state6 (k_{E→E+·T} ν) (reduce_{T→T*F} g_T ν)
        g_F ν                           = state10 (reduce_{T→F} g_T ν)
state6                                  :: 𝒫 r → (MulOp → Factor → 𝒫 r) → 𝒫 r
state6 k_{E→E+T·} k_{T→T·*F} ts         =
  case ts of MULOP ν : tr               → state8 (k_{T→T·*F} ν) tr
             _                          → k_{E→E+T·} ts
state7                                  :: 𝒫 r → (MulOp → Factor → 𝒫 r) → 𝒫 r
state7 k_{E→T·} k_{T→T·*F} ts           =
  case ts of MULOP ν : tr               → state8 (k_{T→T·*F} ν) tr
             _                          → k_{E→T·} ts
state8                                  :: (Factor → 𝒫 r) → 𝒫 r
state8 k_{T→T*·F} ts                    =
  case ts of LPAREN   : tr              → state11 (reduce_{F→(E)} g_F) tr
             IDENT ν  : tr              → state13 (reduce_{F→id} g_F ν) tr
             _                          → ∅
  where g_F ν                           = state9 (k_{T→T*·F} ν)
state9                                  :: 𝒫 r → 𝒫 r
state9 k_{T→T*F·} ts                    = k_{T→T*F·} ts
state10                                 :: 𝒫 r → 𝒫 r
state10 k_{T→F·} ts                     = k_{T→F·} ts
state11                                 :: (Expr → 𝒫 r) → 𝒫 r
state11 k_{F→(·E)} ts                   =
  case ts of LPAREN   : tr              → state11 (reduce_{F→(E)} g_F) tr
             IDENT ν  : tr              → state13 (reduce_{F→id} g_F ν) tr
             _                          → ∅
  where g_E ν                           = state4 (reduce_{E→E+T} g_E ν) (k_{F→(·E)} ν)
        g_T ν                           = state7 (reduce_{E→T} g_E ν) (reduce_{T→T*F} g_T ν)
        g_F ν                           = state10 (reduce_{T→F} g_T ν)
state12                                 :: 𝒫 r → 𝒫 r
state12 k_{F→(E)·} ts                   = k_{F→(E)·} ts
state13                                 :: 𝒫 r → 𝒫 r
state13 k_{F→id·} ts                    = k_{F→id·} ts
```

Note that state_6 and state_7 are identical since the two sets of kernel items do not differ in the strings to the right of the dot.

D. Smaller LALR(1) Expression Parser

Again, parse_E is a deterministic recursive ascent parser for the expression grammar. Compared to the parser in Appendix C, trivial states of the form $\mathsf{state}_i\ \mathsf{k}_q = \mathsf{k}_q$ have been inlined. Furthermore, common actions have been factored out: state_1 is defined in terms of state_5 which in turn depends on state_8.

```
parse_E                          :: 𝒫 Expr
parse_E                          = state_1 accept

state_1                          :: (Expr → 𝒫 r) → 𝒫 r
state_1 k_{E†→ · E$}             = state_5 (reduce_{E→T} g_E)
  where g_E v                    = state_2 (k_{E†→ · E$} v) (reduce_{E→E+T} g_E v)

state_2                          :: 𝒫 r → (AddOp → Term → 𝒫 r) → 𝒫 r
state_2 k_{E†→E · $} k_{E→E · +T} ts =
  case ts of ADDOP v : tr        → state_5 (k_{E→E · +T} v) tr
             EOF     : tr        → k_{E†→E · $} tr
             _                   → ∅

state_4                          :: (AddOp → Term → 𝒫 r) → 𝒫 r → 𝒫 r
state_4 k_{E→E · +T} k_{F→( E · )} ts =
  case ts of RPAREN  : tr        → k_{F→( E · )} tr
             ADDOP v : tr        → state_5 (k_{E→E · +T} v) tr
             _                   → ∅

state_5                          :: (Term → 𝒫 r) → 𝒫 r
state_5 k_{E→E+ · T}             = state_8 (reduce_{T→F} g_T)
  where g_T v                    = state_6 (k_{E→E+ · T} v) (reduce_{T→T*F} g_T v)

state_6                          :: 𝒫 r → (MulOp → Factor → 𝒫 r) → 𝒫 r
state_6 k_{E→E+T ·} k_{T→T · *F} ts =
  case ts of MULOP v : tr        → state_8 (k_{T→T · *F} v) tr
             _                   → k_{E→E+T ·} ts

state_8                          :: (Factor → 𝒫 r) → 𝒫 r
state_8 k_{T→T* · F} ts          =
  case ts of LPAREN  : tr        → state_11 (reduce_{F→( E )} k_{T→T* · F}) tr
             IDENT v : tr        → reduce_{F→id} k_{T→T* · F} v tr
             _                   → ∅

state_11                         :: (Expr → 𝒫 r) → 𝒫 r
state_11 k_{F→( · E )}           = state_5 (reduce_{E→T} g_E)
  where g_E v                    = state_4 (reduce_{E→E+T} g_E v) (k_{F→( · E )} v)
```