Generation of LR Parsers by Partial Evaluation

MICHAEL SPERBER
Universität Tübingen
and
PETER THIEMANN
Universität Freiburg

The combination of modern programming languages and partial evaluation yields new approaches to old problems. In particular, the combination of functional programming and partial evaluation can turn a general parser into a parser generator. We use an inherently functional approach to implement general LR(k) parsers and specialize them with respect to the input grammars using offline partial evaluation. The functional specification of LR parsing yields a concise implementation of the algorithms themselves. Furthermore, we demonstrate the elegance of the functional approach by incorporating on-the-fly attribute evaluation for S-attributed grammars and two schemes for error recovery, which lend themselves to natural and elegant implementation. The parsers require only minor changes to achieve good specialization results. The generated parsers have production quality and match those produced by traditional parser generators in speed and compactness.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative Languages*; D.3.4 [**Programming Languages**]: Processors—*Parsing*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Parsing*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Transformation*; *Program Synthesis*

General Terms: Algorithms, Experimentation, Languages, Performance

Additional Key Words and Phrases: Continuations, functional programming, LR parsing, parser generation, partial evaluation

1. INTRODUCTION

LR parsing [Knuth 1965] is the predominant parsing technique in compiler front ends and other formal language processors. LR parsing is popular because many realistic grammars are immediately amenable to this technique. Also, several LR parser generators are readily available, most notably Yacc [Johnson 1975] which

An earlier version of parts of this article appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995 under the title "The Essence of LR Parsing."

Authors' addresses: M. Sperber, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany; email: sperber@informatik.uni-tuebingen.de; P. Thiemann, Institut für Informatik, Universität Freiburg, Universitätsgelände Flugplatz, D-79110 Freiburg im Breisgau, Germany; email: thiemann@informatik.uni-freiburg.de.

creates LALR parsers.

Unfortunately, the techniques underlying LR parsing are difficult to comprehend. We advocate a fresh approach to creating LR parser generators which ensures understandability and correctness at every step, and leads to extremely efficient parsers:

(1) Write a concise, functional specification of LR parsing which is understandable and easy to prove correct.
(2) Use partial evaluation, an automatic program transformation, to turn the general parser into a generator of efficient parsers. The correctness of the partial evaluator guarantees the correctness of this step.

The reason for the traditional opacity of presentations of LR parsing is mostly due to the implementation methods: most presentations employ a stack automaton [Johnson 1975; Chapman 1987; Sippu and Soisalon-Soininen 1990; Wilhelm and Maurer 1995], and its usual implementation is an interpreter of a transition table. Unfortunately, since the naive table representation of an LR automaton may be prohibitively large, it is necessary to use sparse table representations which in turn require significant algorithmics and coding [Tarjan and Yao 1979; Dencker et al. 1984]. Consequently, the LR parsers resulting from this approach are hard to read, and typically contain numerous obscure performance hacks. This, in turn, makes finding errors and tending to parsing conflicts difficult. Testing is only possible with the generated parsers.

These difficulties are in sharp contrast to LL techniques where grammars translate straightforwardly into readable recursive-descent parsers; this is one of the main advantages of LL techniques as cited by its advocates.

An alternative approach to LR parsing is much more transparent: start from a functional specification of LR(k) parsing and derive the implementation by program calculation [Leermakers 1993]. The resulting formulation of LR parsing turns out to be quite simple. It relies on the *recursive-ascent* technique [Pennello 1986; Roberts 1988], analogous to the recursive-descent method commonly used in LL parsers. Our investigation starts from this inherently functional approach, which does not require an explicit parse table or a parsing stack. By application of well-known program transformation steps, we arrive at an original alternative specification based on continuations. It shares the advantanges of the first approach and is even more concise and elegant.

Recursive ascent results in LR parsers which are considerably simpler and more compact than formulations using a stack automaton. The specifications of the parsing algorithms translate directly into functional programs. These *general parsers* run independently and take both a grammar and an input string as operands. Adding attribute evaluation as well as effective error recovery fits elegantly into the functional framework. General parsers are comparatively easy to write and debug. The interpretive framework also eases writing and debugging of grammars.

Consequently, using an inherently functional approach as well as a functional programming language to implement LR parsing yields new insights into the methodology of software development, and suggests that functional programming offers fresh ideas for many old software construction problems.

Starting from a general parser, a *partial evaluator* can, given an input grammar and lookahead, automatically construct efficient specialized parsers. One particularly simple partial-evaluation technique, *offline partial evaluation*, suffices for this task. The same technique can automatically generate efficient standalone parser generators. The resulting parsers are simple in structure, consisting of mutually recursive procedures that encode parser states. To achieve good results with offline partial evaluation, many realistic programs require slight changes, so-called *binding-time improvements.* Our parsers are no exception. However, the techniques employed are fairly standard, and the improvements do not disrupt the structure of the original parsers.

The parser implementations are written in Scheme [IEEE 1991; Kelsey et al. 1998] which makes for particularly concise and readable programs. It also facilitates the use of existing partial-evaluation systems for Scheme. We have used two different systems for developing the parser generators: Bondorf's Similix [Bondorf 1991] for the initial experiments, and Thiemann's PGG system [Thiemann 1996a] for the production version. Both the PGG system as well as the parser code are publicly available on the Internet [Sperber and Thiemann 1999; Thiemann 1999].

In addition to the parsing algorithms, we have implemented attribute evaluation and two styles of error recovery—the method of Yacc [Johnson 1975], which depends on user annotations to the input grammar, and Röhrich's automatic backtrack-free mechanism [Röhrich 1980]. This demonstrates the feasibility of our approach.

Compared to real-world parser generators, the performance of the generated parsers is competitive. When compiled with a high-performance Scheme implementation, they are about as fast as parsers generated by Bison [Donnelly and Stallman 1995].

Overview The article is organized as follows: the first three sections introduce some notational preliminaries and two approaches to functional LR parsing: the so-called *direct-style* approach (Section 3) and the *continuation-based* approach (Section 4). The presentation includes straightforward implementations in Scheme. The following Section 5 is an introduction to offline partial evaluation.

Section 6 describes the specialization of the parsing algorithm in direct style, specifically the binding-time improvements. Moving on to the continuation-based approach, Section 7 describes the additional problems that had to be solved to obtain satisfactory results. A refinement of the checking of the lookahead, which improves both styles of parsers, is the topic of Section 8.

Section 9 describes the implementation of attribute evaluation and error recovery for the continuation-based implementation. Section 10 gives the results of practical experiments; Section 11 discusses related work.

For concreteness, all concepts are explained and implemented using the programming language Scheme [Kelsey et al. 1998]. The conciseness of the language enables us to include actual code in the article.

2. NOTATIONAL PRELIMINARIES

This section reviews notation for sequences and for context-free grammars. It also introduces the *bunch notation* [Leermakers 1993] for expressing nondeterministic algorithms.

2.1 Sequences

For a set A, A^* denotes the set of finite sequences of elements of A with ε denoting the empty sequence. For some $\xi = a_1 \dots a_n \in A^*$ let $|\xi| := n$ be the length of ξ. The k-prefix of ξ, $\xi_{|k}$, is $a_1 \dots a_j$ where $j = \min(k, n)$.

2.2 Context-Free Grammars

A *start-separated context-free grammar* (or just *grammar*) is a tuple $G = (N, T, P, S)$. N is the set of *nonterminals*, T the set of *terminals* (disjoint from N), $S \in N$ the *start symbol*, $V = T \cup N$ the set of *grammar symbols*. P is the set of productions; productions have the form $A \rightarrow \alpha$ for a nonterminal A and a sequence α of grammar symbols. An *ε-production* has the form $A \rightarrow \varepsilon$. There is exactly one production in P with S as its left-hand side. It has the form $S \rightarrow A$ where $A \in N$.

Some letters denote elements of certain sets by convention:

$$\begin{aligned} A, B, C, E &\in N \\ \xi, \rho, \tau &\in T^* \\ x, y, z &\in T \\ \alpha, \beta, \gamma, \delta, \nu, \mu &\in V^* \\ X, Y, Z &\in V \end{aligned}$$

G induces the *derives relation* $\Rightarrow$ on V^* with

$$\alpha \Rightarrow \beta \ :\Leftrightarrow \ \alpha = \delta A \gamma \ \wedge \ \beta = \delta \mu \gamma \ \wedge \ A \rightarrow \mu$$

where $A \rightarrow \mu$ stands for $\exists p \in P. p = A \rightarrow \mu$ and where δ, A, γ, and μ are implicitly existentially quantified on the right side of the $\Leftrightarrow$. The reflexive and transitive closure of $\Rightarrow$ is $\overset{*}{\Rightarrow}$. A *derivation* from α_0 to α_n is a sequence $\alpha_0, \alpha_1, \dots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq n$. A *sentential form* is a sequence appearing in a derivation beginning with the start symbol. The *leftmost-symbol rewriting* $\underset{\ell}{\Rightarrow}$ is a relation defined as

$$B\alpha \underset{\ell}{\Rightarrow} \delta\alpha \ :\Leftrightarrow \ B \rightarrow \delta \wedge \delta \neq \varepsilon$$

with reflexive and transitive closure $\underset{\ell}{\overset{*}{\Rightarrow}}$.

An *LR(k) item* (or just *item*) is a triple consisting of a production $A \rightarrow \mu$, a position $0 \leq j \leq |\mu|$ within its right-hand side, and a terminal string of length $\leq k$—the *lookahead*. An item is written as $A \rightarrow \alpha \cdot \beta \ (\rho)$ where the dot indicates the position, and ρ is the lookahead. When the lookahead is irrelevant (or $k = 0$), it is omitted. A *kernel item* has the form $A \rightarrow \alpha \cdot \beta \ (\rho)$ with $|\alpha| > 0$. A *predict item* has the form $A \rightarrow \cdot \alpha \ (\rho)$.

An *LR(k) state* (or just *state*) q is a nonempty set of LR(k) items.

In the following, all productions, items, and states implicitly range over the productions, items, and states of some arbitrary, but fixed, context-free grammar. Also, all derivations and rewrite steps are induced by this grammar.

2.3 Bunches

Bunches are a notational convenience for expressing nondeterministic algorithms in a more readable way than a set-based notation [Leermakers 1993].

A *bunch* denotes a nondeterministic choice of values. Any normal expression denotes a singleton bunch with just one value. If a and b are bunches, then $a \mid b$ is a bunch consisting of the values of a and b. An empty bunch is said to fail and is therefore denoted by *fail*. In other words, $\mid$ is a nondeterministic choice operator with unit *fail*. It is associative, commutative, and idempotent. A function can only be applied to a value, the member of a singleton bunch. If the argument contains a choice it must be distributed, first, i.e., $f(\mathit{fail}) = \mathit{fail}$ and $f(a \mid b) = f(a) \mid f(b)$.

The *guarded expression* $x \leftarrow a \rhd b$ stands for the bunch of all values of b where x ranges over all values of a. Hence, it behaves like the set comprehension $\bigcup\{b \mid x \in a\}$. When x does not occur in b, $a \rhd b$ is a valid shorthand. In this case, if the guard a fails, the entire expression fails; otherwise, if $a \neq \mathit{fail}$, $a \rhd b = b$. Again, this is the behavior of the above set comprehension for $a = \emptyset$ (the translation of *fail* to sets). It is convenient to admit logical predicates as guards. A predicate denotes the singleton bunch *true* if it is satisfied; otherwise it denotes *fail*. For further convenience, guards sometimes use an informal pattern-matching notation. Such a guarded expression also fails if the pattern does not match the value provided.

3. DIRECT-STYLE FUNCTIONAL LR PARSING

This section introduces one approach to functional LR parsing. The next section will transform this approach into an even more elegant formulation, which is based on continuations. To contrast the two, the approach presented in this section is called the *direct-style* approach.

Roughly, a parser for a context-free grammar is a function that maps a sequence of terminals ξ to a derivation from the start symbol to ξ. An LR parser implements this function using an automaton operating on LR(k) states. When the automaton is in state q, an item $A \rightarrow \alpha \cdot \beta \in q$ means that the parser has processed a suffix derivable from α in the input and is looking for something derivable from β next.

A parser state distinguishes between three cases. Either $\beta = x\delta$ where x is also the next terminal in the input. In that case, the parser can *shift on* x, moving to a state that contains $A \rightarrow \alpha x \cdot \delta$. Otherwise, if $\beta = \varepsilon$, the parser has seen input derivable from A. Now, either $\alpha = \varepsilon$ so that the parser can directly shift on A or $\alpha \neq \varepsilon$ in which case the parser must return to the state q introducing that production. Hence, q must contain the item $A \rightarrow \cdot\alpha$ and items of the form $B \rightarrow \gamma \cdot A\delta$. In state q (which corresponds to the third case $\beta = A\delta$), the parser can shift on A, effectively *reducing* with the production $A \rightarrow \alpha$ and making it part of the generated derivation.

3.1 Specifying Direct-Style LR Parsing

A functional LR parser is a set of mutually recursive functions $[q]$, each of which corresponds to an LR state q. For each $A \rightarrow \alpha \cdot \beta \in q$ and for each derivation from β to ξ there is a representative value $(A \rightarrow \alpha \cdot \beta, \xi')$ in the bunch $[q](\xi\xi')$. For each derivation found the bunch contains the item from which it originated and the rest of the input string not yet parsed. Hence, the following specification formalizes LR parsing [Leermakers 1993]:

$$\begin{aligned}[q](x_1 x_2 \ldots x_n) = {} & \big(A \rightarrow \alpha \cdot \beta \in q \wedge \beta \overset{*}{\Rightarrow} x_1 \ldots x_j\big) \\ & \rhd (A \rightarrow \alpha \cdot \beta, x_{j+1} \ldots x_n)\end{aligned}$$

The initial state of an LR parser is q_0 with $q_0 = \{S \rightarrow \cdot A\}$. $S \overset{*}{\Rightarrow} \xi$ holds if and only if $(S \rightarrow \cdot A, \varepsilon)$ occurs in the bunch produced by $[q_0](\xi)$.

A few auxiliary definitions are necessary to cast the specification into an algorithm. The *first*$_k$ function computes lookahead information:

$$\mathit{first}_k(\alpha) := \left\{\xi_{|k} \mid \alpha \overset{*}{\Rightarrow} \xi\right\}$$

Each state q comes with a set of *predict items*

$$\begin{aligned}\mathit{predict}(q) := \big\{B \rightarrow \cdot\nu\ (\tau) \mid\ & A \rightarrow \alpha \cdot \beta\ (\rho) \Downarrow^+ B \rightarrow \cdot\nu\ (\tau) \\ & \text{for } A \rightarrow \alpha \cdot \beta\ (\rho) \in q\big\}\end{aligned}$$

where $\Downarrow^+$ is the transitive closure of the relation $\Downarrow$ with exactly the following pairings:

$$A \rightarrow \alpha \cdot B\beta\ (\rho) \Downarrow B \rightarrow \cdot\delta\ (\tau) \quad \text{for all } \tau \in \mathit{first}_k(\beta\rho)$$

The predict items of a state q are predictions on what derivations the parser may enter next when in state q. The elements of *predict*(q) are exactly those at the end of leftmost-symbol rewritings starting from items in q. The union of q and *predict*(q) is called the *closure* of q, denoted by

$$\mathit{closure}(q) := q \cup \mathit{predict}(q).$$

Repeated application of *goto* generates all other states. For a state q and a grammar symbol X:

$$\mathit{goto}(q, X) := \{A \rightarrow \alpha X \cdot \beta\ (\rho) \mid A \rightarrow \alpha \cdot X\beta\ (\rho) \in \mathit{closure}(q)\}$$

The general parser involves an auxiliary function $\overline{[q]}$ for each state q. An invocation $\overline{[q]}(X, x_1 \ldots x_n)$ means that a grammar symbol X or a string derived from it has just been seen in the input and removed from it. The specification is

$$\begin{aligned}&\overline{[q]}\ (X, x_1 \ldots x_n) = \\ &\quad (A \rightarrow \alpha \cdot \beta \in q \wedge \beta \underset{\ell}{\overset{*}{\Rightarrow}} X\gamma \wedge \gamma \overset{*}{\Rightarrow} x_1 \ldots x_j) \\ &\quad\ \ \rhd (A \rightarrow \alpha \cdot \beta, x_{j+1} \ldots x_n)\end{aligned}$$

Thus, $\overline{[q]}$ is called whenever the functional equivalent of a "shift" action happens. Its implementation calls $[\mathit{goto}(q, X)]$ recursively and either returns the result (for a kernel item of the form $A \rightarrow \alpha \cdot \beta$ where $|\alpha| > 0$) or shifts the left-hand-side nonterminal for a predict item of the form $C \rightarrow \cdot\alpha$, implemented as a recursive call of $\overline{[q]}$.

3.2 Formulating Direct-Style LR Parsing

Figure 1 gives an "implementation" of the above functions [Leermakers 1993]. The implementation results from the specification by straightforward calculation. It includes testing the k-lookahead. Hence it implements a nondeterministic LR(k) parsing algorithm. If the grammar is such that the parser is deterministic, the grammar is called *LR(k)*. This coincides with more traditional definitions of the term [Knuth 1965; Chapman 1987; Sippu and Soisalon-Soininen 1990; Wilhelm and Maurer 1995].

In the implementation, the function $[q]$ first tries to shift on the next terminal symbol x. If there is an ε-production $B \rightarrow \cdot$ in q, then $[q]$ shifts on B. Finally, if

$$\begin{array}{lll}
[q](\xi) := & x\xi' \leftarrow \xi & \triangleright\ \overline{[q]}(x, \xi') \\
& \mid\ B \rightarrow \cdot\ (\rho) \in \mathit{predict}(q) \wedge (\xi)_{|k} = \rho & \triangleright\ \overline{[q]}(B, \xi) \\
& \mid\ A \rightarrow \alpha \cdot\ \ (\rho) \in q \wedge (\xi)_{|k} = \rho & \triangleright\ (A \rightarrow \alpha\cdot, \xi) \\
& & \\
\overline{[q]}(X, \xi) := & A \rightarrow \alpha \cdot X\gamma \in q\ \wedge & \\
& (A \rightarrow \alpha X \cdot \gamma, \tau) \leftarrow [\mathit{goto}(q, X)]\,(\xi) & \triangleright\ (A \rightarrow \alpha \cdot X\gamma, \tau) \\
& \mid\ C \rightarrow \cdot X\delta \in \mathit{predict}(q)\ \wedge & \\
& (C \rightarrow X \cdot \delta, \tau) \leftarrow [\mathit{goto}(q, X)]\,(\xi) & \triangleright\ \overline{[q]}(C, \tau)
\end{array}$$

Fig. 1. Functional LR(k) parser, direct-style version.

q contains an item $A \rightarrow \alpha\cdot$ and, therefore, a reduction is possible, and the parser returns that item along with the part of the input string not yet seen.

$\overline{[q]}$ first calls the function associated with $\mathit{goto}(q, X)$, which returns an item and a rest string. When the parser has arrived at a state containing an item $A \rightarrow \alpha\cdot$, it returns through $|\alpha| - 1$ function invocations, moving the dot back one symbol on each return: $A \rightarrow \alpha X \cdot \gamma$ returned by $[\mathit{goto}(q, X)]$ becomes $A \rightarrow \alpha \cdot X\gamma$. The second alternative in $\overline{[q]}$ is considered when the dot has arrived at the front of a right-hand side $C \rightarrow \cdot X\delta$ by the process just described. In that case, $\overline{[q]}$ shifts on the corresponding left-hand side C.

This technique for implementing LR parsing is called *recursive ascent* because information about a production selected for inclusion in the derivation is passed on returning from a parsing function, thereby ascending in the function call graph. This is in contrast to the recursive-descent technique for implementing LL parsing, where the parsing function selects a production to which it then makes a call, thereby descending in the call graph.

Returning items from function calls may seem prohibitively expensive. In practice, it is sufficient to return the left-hand side of the production and an integer counting the number of levels left to return through. Our implementation of the specification exploits this property.

Finally, the functional LR parsers have a close relation to the well-known stack-based implementations: The "run-time stack" of the procedure calls corresponds to the parsing stack of traditional expositions of LR parsing.

3.3 Implementing Direct-Style LR Parsing

The formal specification of functional LR parsing gives rise to a naive working implementation of a general LR parser. Assuming an LR(k) input grammar, the LR(k) parsing algorithm in Figure 1 becomes deterministic and thus transliterates easily into a functional program which is shown in Figure 2.

The procedure `ds-parse` implements the function $[\cdot]$. It accepts a grammar `grammar`, the number of lookahead characters `k`, a function `compute-closure` that computes the closure of a state, a set of LR(k) items `state`, and the input `input`. The input is represented by a stream of pairs. Each pair contains a terminal as its first component and an attribute value as its second component. (Attribute values are only relevant for attribute evaluation discussed in Section 9.2.) `Compute-closure` is a parameter so that the parser is sufficiently general to also

```
(define (ds-parse grammar k compute-closure state input)
  (let ((closure (compute-closure state grammar k)))
    (cond
     ((and (not (stream-empty? input))
           (member (car (stream-car input))
                   (next-terminals closure grammar)))
      (ds-parse-bar grammar k compute-closure closure
                    (car (stream-car input)) (stream-cdr input)))
     ((find-lookahead-item (accept closure) k input)
      => (lambda (item)
           (let ((rhs-length (length (item-rhs item)))
                 (lhs (item-lhs item)))
             (if (zero? rhs-length)
                 (ds-parse-bar grammar k compute-closure
                               state lhs input)
                 (values lhs rhs-length input)))))
     (else (error "parse error")))))

(define (ds-parse-bar grammar k compute-closure state symbol input)
  (let ((closure (compute-closure state grammar k)))
    (call-with-values
     (lambda ()
       (ds-parse grammar k compute-closure
                 (goto closure symbol) input))
     (lambda (lhs dot input)
       (cond
        ((> dot 1)
         (values lhs (- dot 1) input))
        ((and (initial? state grammar)
              (equal? (grammar-start grammar) lhs))
         (if (stream-empty? input)
             'accept
             (error "parse error")))
        (else
         (ds-parse-bar grammar k compute-closure
                       state lhs input)))))))
```

Fig. 2. Scheme implementation of direct-style functional LR(k) parser.

perform SLR or LALR parsing; Section 9.1 provides details.

The $[\cdot]$ function returns three values—the left-hand side of the production which has been reduced, the position of the dot in the right-hand side of that production, and the remaining input. The function `find-lookahead-item` selects an item from the current state matching the lookahead in the current input. (The `cond` conditional form passes the return value of `find-lookahead-item` to the function in the `=>` clause if it is not `#f`.) `Accept` extracts all items of the form $A \to \alpha\cdot$ from the closure of `state`. `Find-lookahead-item` either returns an item matching the lookahead or `#f` if no such item exists. Sections 6.1.2 and 8 discuss implementation strategies for `find-lookahead-item`.

The implementation employs Scheme's support for functions returning multiple values [Kelsey et al. 1998]. The values procedure constructs a multiple return value from its arguments. Its counterpart, call-with-values, is for processing multiple values returned from a function: it calls the function of no arguments which is its first parameter, and subsequently calls the function which is its second parameter, passing the return values of the first function as arguments.

The ds-parse-bar procedure is the implementation of $\overline{[\cdot]}$. It first computes the next state using the goto function and applies ds-parse to obtain lhs, the left-hand side of the production which has been reduced, dot, the position of the dot in the right-hand side of that production, and input, the remaining input.

The position of the dot in the returned item guides further actions. If dot is not zero yet, the parser must move further backward. In this case, ds-parse-bar terminates with the dot shifted by one symbol to the left. The left-hand side lhs and the remaining input input remain unchanged. Hence, the result is (values lhs (- dot 1) input). Otherwise, if dot has reached zero, all symbols of the right-hand side of the reduced production have already been popped. Then, ds-parse-bar is tail-called on the current state. It shifts the symbol lhs and consumes the remaining input input.

Ds-parse-bar also performs checking for the end of input which is only allowed in the initial state of the grammar. (This checking is not present in the specification which implicitly assumes the input is delimited by k end-of-input symbols.) Initial? checks if a state is the initial state.

Most of the auxiliary functions such as those for constructing and accessing items (item-lhs, item-rhs) as well as for accessing grammars (grammar-start), etc, are trivial. Only compute-closure is slightly more involved as it constructs a transitive closure.

4. CONTINUATION-BASED LR PARSING

The direct-style functional approach is a feasible way to implement LR parsing. However, it is possible to do better by expressing LR parsing with continuations [Sperber 1994]. The continuation-based parser is the result of applying a sequence of well-known transformation steps to the direct-style specification. The first step transforms the direct-style parser into continuation-passing style as in Leermakers [1993]. The next step changes the representation of continuations in the resulting parser using closure conversion [Reynolds 1972]. The final step is a subtle representation change of the parsing state in the continuation closures.

The resulting continuation-based parser is more concise than the direct-style parser and just as amenable to effective specialization. In addition, it allows for a natural and elegant implementation of attribute evaluation and error recovery (see Section 9). It is also suitable for further improvements beyond the scope of this article, notably an extended model of attribute evaluation [Sperber 1994], which would be cumbersome to implement in the direct-style version.

4.1 Transformation to Continuation-Passing Style

Figure 3 shows the result of transforming the direct-style specification of LR parsing from Figure 1 to continuation-passing style [Reynolds 1972; Plotkin 1975; Fischer 1993; Danvy and Filinski 1992]. Both functions [q] and $\overline{[q]}$ receive an additional continuation parameter c, which maps a pair of an LR(k) item and an input string to

$$
\begin{array}{lll}
[q](\xi, c) := & x\xi' \leftarrow \xi & \rhd\ \overline{[q]}(x, \xi', c) \\
 & \mid\ B \rightarrow \cdot\ (\rho) \in \mathit{predict}(q) \wedge (\xi)_{|k} = \rho & \rhd\ \overline{[q]}(B, \xi, c) \\
 & \mid\ A \rightarrow \alpha \cdot\ \ (\rho) \in q \wedge (\xi)_{|k} = \rho & \rhd\ c(A \rightarrow \alpha \cdot, \xi) \\
\overline{[q]}(X, \xi, c) := & [\mathit{goto}(q, X)](\xi, d(q, c)) & \\
d(q, c)(A \rightarrow \alpha X \cdot \gamma, \tau) := & A \rightarrow \alpha \cdot X\gamma \in q & \rhd\ c(A \rightarrow \alpha \cdot X\gamma, \tau) \\
 & \mid\ A \rightarrow \alpha \cdot X\gamma \in \mathit{predict}(q) & \rhd\ \overline{[q]}(A, \tau, c)
\end{array}
$$

Fig. 3. Direct-style functional LR(k) parser after transformation to continuation-passing style.

$$
\begin{array}{lll}
[q](\xi, c) := & x\xi' \leftarrow \xi & \rhd\ [\mathit{goto}(q, x)](\xi', d(q, c)) \\
 & \mid\ B \rightarrow \cdot\ (\rho) \in \mathit{predict}(q) \wedge (\xi)_{|k} = \rho & \rhd\ [\mathit{goto}(q, B)](\xi, d(q, c)) \\
 & \mid\ A \rightarrow \alpha \cdot\ \ (\rho) \in q \wedge (\xi)_{|k} = \rho & \rhd\ a(c, (A \rightarrow \alpha \cdot, \xi)) \\
a(d(q, c), (A \rightarrow \alpha X \cdot \gamma, \tau)) := & A \rightarrow \alpha \cdot X\gamma \in q & \rhd\ a(c, (A \rightarrow \alpha \cdot X\gamma, \tau)) \\
 & \mid\ A \rightarrow \alpha \cdot X\gamma \in \mathit{predict}(q) & \rhd\ [\mathit{goto}(q, A)](\tau, d(q, c))
\end{array}
$$

Fig. 4. Continuation-passing-style LR(k) parser after inlining and closure conversion.

a final answer. Following Leermakers [1993], a function d creates the continuation. It has been lifted to the toplevel by introducing the free variables q and c as parameters. This will prove convenient for subsequent transformation steps.

An obvious transformation is inlining $\overline{[q]}$ into the definition of $[q]$. This moves the case analysis into the continuation.

4.2 Applying Closure Conversion

Closure conversion [Reynolds 1972] replaces the construction of a first-class function by the construction of a data structure (a *closure*) which contains a tag identifying the body of the function and the values of its free variables. In addition, the transformation replaces function application $c(v)$ by a call to a function $a(c, v)$. The function a performs a case analysis of c's tag and dispatches to the body of the corresponding function after putting the values of the free variables in place.

In the parser, the only function closure arising is the closure of the continuation $d(q, c)$ where q and c are the required values of the free variables. Hence, a simple reinterpretation of d as the constructor tag for its closure yields the formulation in Figure 4.

4.3 Changing the Representation

The final transformation step changes the closure representation by including the function $[\mathit{goto}(q, A)]$ called in the second branch of a in the closure. To this end, the transformation introduces the function $f(q, k)$ that performs all shift actions on nonterminal symbols. It replaces the state q in the continuation closure. Figure 5 shows the resulting parser.

It is easy to see that the representation of a continuation is a list of functions $f(q, c)$ for varying q and c. Consequently, the function a is just a list-indexing function where the index is the position of the dot in the item argument. Finally,

$$
\begin{array}{llll}
[q](\xi, c) := & & x\xi' \leftarrow \xi & \rhd\ [\mathit{goto}(q, x)](\xi', d(f(q, c), c)) \\
& \mid & B \rightarrow \cdot\ (\rho) \in \mathit{predict}(q) \wedge (\xi)_{|k} = \rho & \rhd\ [\mathit{goto}(q, B)](\xi, d(f(q, c), c)) \\
& \mid & A \rightarrow \alpha \cdot\ (\rho) \in q \wedge (\xi)_{|k} = \rho & \rhd\ a(c, (A \rightarrow \alpha \cdot, \xi)) \\
a(d(f, c), (A \rightarrow \alpha X \cdot \gamma, \tau)) := & & \alpha \neq \varepsilon & \rhd\ a(c, (A \rightarrow \alpha \cdot X\gamma, \tau)) \\
& \mid & \alpha = \varepsilon & \rhd\ f(A, \tau) \\
f(q, c)(A, \tau) := & & A \rightarrow \cdot X\gamma \in \mathit{predict}(q) & \rhd\ [\mathit{goto}(q, A)](\tau, c)
\end{array}
$$

Fig. 5. Continuation-passing-style LR(k) after change in closure representation.

$$
\begin{array}{l}
[q](\xi, c_1, \ldots, c_{\mathit{nactive}(q)}) := \\
\quad \textbf{letrec} \\
\quad\quad c_0(X, \xi) = [\mathit{goto}(q, X)]\ (\xi, c_0, c_1, \ldots, c_{\mathit{nactive}(\mathit{goto}(q, X))-1}) \\
\quad \textbf{in} \\
\quad\quad A \rightarrow \alpha \cdot\ (\rho) \in \mathit{closure}(q) \wedge (\xi)_{|k} = \rho\ \ \rhd\ \ c_{|\alpha|}(A, \xi) \\
\quad \mid\ \ x\xi' \leftarrow \xi \wedge x \in \mathit{nextterm}(q) \qquad\qquad \rhd\ \ c_0(x, \xi')
\end{array}
$$

Fig. 6. Functional LR(k) parser, continuation-based version.

the parser $[q](\xi, c)$ accesses its continuation at most at the maximal position of the dot in an item in q. Formally, this is the *number of active symbols* of state q, *nactive*(q):

$$\mathit{nactive}(q) := \max\{|\alpha| : A \rightarrow \alpha \cdot \beta \in q\}$$

When the parser is in state q, then *nactive*(q) is the maximal number of states through which the parser may have to go back when it reduces by a production in q. Therefore, as a further improvement, each call to $[q]$ can prune c to length *nactive*(q).

4.4 Formulating Continuation-Based LR Parsing

Two further transformation steps yield Sperber's LR parser [Sperber 1994]. Sperber passes the functions in the list as separate arguments so that each access—and hence each reduction step—takes unit time. The crucial insight here is that only the first *nactive*(q) elements of the list are accessed, as discussed above. This change of representation facilitates merging all cases where *goto* is called into one line of code.

A further definition is necessary to specify the resulting code concisely. The function *nextterm* maps an LR state q to the set of all terminals on which q might perform a shift action:

$$\mathit{nextterm}(q) := \{x \mid A \rightarrow \alpha \cdot x\beta \in \mathit{closure}(q)\}$$

With this definition, Figure 6 shows the resulting specification.

The locally created continuation c_0 performs a state transition. The invocation of c_0 with a grammar symbol X means "return to state q and shift on X." The parameter c_1 is the continuation created by the previous state, c_2 the one created by the state two symbols back, and so on. With the continuations in place, the

```
(define (cps-parse grammar k compute-closure
                   state continuations input)
  (let ((closure (compute-closure state)))

    (define (c0 symbol input)
      (cond
       ((not (and (initial? state grammar)
                  (equal? (grammar-start grammar) symbol)))
        (cps-parse grammar k compute-closure
                   (goto closure symbol)
                   (cons c0
                         (take (- (nactive (goto closure symbol)) 1)
                               continuations))
                   input))
       ((stream-empty? input) 'accept)
       (else 'error)))

    (cond
     ((and (not (stream-empty? input))
           (member (car (stream-car input))
                   (next-terminals closure grammar)))
      (c0 (car (stream-car input)) (stream-cdr input)))
     ((find-lookahead-item (accept closure) k input)
      => (lambda (item)
           ((list-ref (cons c0 continuations)
                      (length (item-rhs item)))
            (item-lhs item) input)))
     (else 'error))))
```

Fig. 7. Scheme implementation of continuation-based parser.

actual parser dispatch has only two cases: when the parser has seen the complete right-hand side of a production and the lookahead matches, it goes back the number of steps indicated by the length of the right-hand side. It does so by directly calling the continuation belonging to the destination state instead of shifting the dot back through multiple procedure invocations. It is sure to reach a state which can shift on the left-hand side of the production. Moreover, when the parser sees a terminal matching the next terminal of an item, it shifts on that terminal by calling the continuation created in the current state.

4.5 Implementating Continuation-Based LR Parsing

Just as with the direct-style parser, the implementation is deterministic when given an LR(k) grammar. Figure 7 shows the transliteration of the above specification. The parameter `continuations` is a list containing the continuations $c_1, \ldots, c_{nactive(\texttt{state})}$ from the specification. The `take` function takes two arguments, n and l, and extracts the first n elements of the list l. Otherwise, the code is a direct transliteration of the formulation in Figure 6. The only thing it adds is proper handling for the end of the input and a preference of shifting over reducing if both are possible.

5. PRAGMATICS OF OFFLINE PARTIAL EVALUATION

This section provides some general background on partial evaluation. Readers familiar with the field may want to skip it. In particular, the section gives a high-level overview of what partial evaluation is, a brief introduction into the workings of offline partial evaluators, an overview over some of the binding-time improvements necessary for making programs amenable to offline partial evaluation, and a description of performing partial evaluation with generating extensions.

5.1 Preliminaries

Partial evaluation is a specialization technique based on aggressive constant propagation: if parts of the input of a *subject program* are known at compile time, a partial evaluator propagates this *static* input to generate a *specialized program.* The specialized program takes the remaining, *dynamic* parts of the input as parameters and produces the same results as the subject program applied to the complete input.

For example, consider a program text p with semantics $\llbracket \mathrm{p} \rrbracket$, taking two inputs in_1 and in_2. Suppose p terminates producing an output *out*:

$$out := \llbracket \mathrm{p} \rrbracket(in_1, in_2)$$

This situation provides opportunity for partial evaluation, by first running a partial evaluator *pe* on p and in_1 (which supposedly terminates):

$$\mathrm{p}_{in_1} := pe(\mathrm{p}, in_1)$$

The resulting specialized program p_{in_1} must terminate and produce the same output *out* when applied to in_2:

$$out = \llbracket \mathrm{p}_{in_1} \rrbracket(in_2)$$

5.2 Offline Partial Evaluation

Offline partial evaluation consists of two stages, a *binding-time analysis* and a *static reducer.* The binding-time analysis annotates all expressions of a subject program with a *binding time*—either as executable at partial-evaluation time (*static*) or deferred to run time (*dynamic*), usually starting from a user-supplied annotation of the inputs of the program.

A prominent example for this kind of scenario is, of course, a general parser which essentially accepts a static grammar and a dynamic input as arguments. Another popular, simpler example for partial evaluation is the standard `append` function that concatenates two lists. With its first input annotated as static and its second input dynamic, the output of the binding-time analysis may look as follows, with the dynamic operations underlined:

```
(define (append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

This type of annotation actually implies a *monovariant* binding-time analysis—each expression receives one (and only one) annotation, and thus appears in only one binding-time context. This sometimes forces the analysis to annotate an expression as dynamic even though it may be static in certain situations during specialization.

The static reducer processes the annotated program and the static inputs to generate a specialized program, reducing the static parts and rebuilding the dynamic parts.

For the `append` example, this means that the reducer is always able to evaluate the nonunderlined parts, and whenever it encounters an underlined construct, the construct shows up in the specialized program. Hence, the specialized program will always consist of a chain of `cons` applications, ending with `l2`. For instance, if the reducer specializes `append` with `l1` = `'(foo bar baz)`, the following specialized program results:

```
(define (append_1 l2)
  (cons 'foo (cons 'bar (cons 'baz l2))))
```

By default, the static reducer unfolds all procedure applications, as with the recursive call to `append` above. For some recursive procedure calls unfolding might result in infinite specialization. Therefore, the static reducer must generate specialized procedures for these calls to preserve the recursion. The expressions for which the static reducer generates such procedures are called *memoization points*. When the static reducer encounters such a memoization point, it does not unfold but generates a procedure call instead. If it has already encountered the same memoization point with the same static components of its free variables, the generated call refers to a previously generated procedure. Otherwise, the reducer must generate an appropriate specialized procedure.

With the `append` example, the binding-time analysis must insert a memoization point at the recursive call if `l1` is dynamic, so as to prevent infinite unfolding.

5.3 Binding-Time Improvements

Every program transformation which enables the binding-time analysis to mark more expressions as static is a *binding-time improvement*. These improvements are sometimes necessary to ensure good specialization results.

5.3.1 *The Trick.* "The Trick" is a well-known binding-time improvement [Jones et al. 1993]. It applies to a dynamic value `d` which is known to belong to a finite set `F` of static values. The Trick amounts to choosing an appropriate context C of `d` and replacing C[`d`] by a loop:

```
C[d] ⇒ foreach s ∈ F do
           if s = d then C[s] else continue
```

Some partial evaluation systems will automatically select a context C and propagate it to the static value inside the loop [Lawall and Danvy 1994]. With such context propagation in place, we only need to replace `d` by a loop to get the effect of The Trick:

```
d ⇒ foreach s ∈ F do
        if s = d then s else continue
```

To achieve context propagation, the binding-time analysis must *not* insert a memoization point at the dynamic conditional `s` = `d`. The directive `define-without-`

`memoization` tells the PGG system to do so. The loop translates directly into a Scheme procedure which returns a static version of the dynamic parameter `element` if it occurs in the static list `set`:

```
(define-without-memoization (maybe-the-member element set)
  (let loop ((set set))
    (cond
     ((null? set) #f)
     ((equal? element (car set)) (car set))
     (else (loop (cdr set))))))
```

The above code returns `#f` in case `element` does not occur in `set`. If, however, `element` is known to be in `set`, a slightly modified version of `maybe-the-member` can exploit that knowledge and omit the last comparison.

```
(define-without-memoization (the-member element set)
  (let loop ((set set))
    (cond
     ((null? (cdr set)) (car set))
     ((equal? element (car set)) (car set))
     (else (loop (cdr set))))))
```

5.3.2 *Polyvariant Expansion.* Monovariant binding-time analyses often mark expressions as dynamic even though they are static in some contexts. Consider the following code fragment, assuming that `main` will be called with `x` static and `y` dynamic:

```
(define (main x y)
  (+ (f x) (f y)))
(define (f z)
  (+ z 1))
```

In principle, the specializer could perform the addition on `x`, but not on `y`. However, since the monovariant binding-time analysis provides only one annotation for the argument of `f`, it must annotate it as dynamic for all situations—an overly conservative choice which makes for unsatisfactory specialization; namely for `x = 5` the specialized program is

```
(define (main-5 y)
  (+ (+ 5 1) (+ y 1)))
```

It is possible to get around the problem by providing two versions of `f`—one for each binding-time pattern:

```
(define (main x y)
  (+ (f-s x) (f-d y)))
(define (f-s z)
  (+ z 1))
(define (f-d z)
  (+ z 1))
```

Specialization of this program produces the desired result:

```
(define (main-5 y)
  (+ 6 (+ y 1)))
```

5.4 Generating Extensions

An attractive feature of partial evaluation is the ability to construct *generating extensions*. A generating extension for the general scenario of Section 5.1 above—a program p with two inputs in_1 and in_2—is a program $\mathtt{p}_{gen}$ which accepts the static input in_1 of p and produces the specialized program $\mathtt{p}_{in_1}$:

$$\begin{aligned}\mathtt{p}_{gen} &= pgg(\mathtt{p})\\ \mathtt{p}_{in_1} &= [\![\mathtt{p}_{gen}]\!](in_1)\end{aligned}$$

The *pgg* function that produces the generating extension is a *program-generator generator* (*PGG*) to p. (Actually, it is possible to generate a PGG automatically from a "normal" partial evaluator by a technique called double self-application [Futamura 1971; Turchin 1979; Jones et al. 1993]. However, it is more effective to write the PGG directly.) A PGG is also called a *compiler generator* (*cogen*) because it can turn an interpreter into a compiler. Specialization with generating extensions is attractive because it is considerably faster than specialization with a normal, interpretive partial evaluator [Bondorf 1991].

Applied to the parsing scenario, a PGG turns a general parser into a parser generator. In this article, we use Thiemann's PGG system [Thiemann 1996a] for the implementation.

6. GENERATING EFFICIENT DIRECT-STYLE LR PARSERS

The implementation of direct-style parsing presented in Section 3.3 is not directly amenable to efficient offline partial evaluation; a few improvements are necessary.

6.1 Improving Binding Times

Three applications of The Trick are sufficient to specialize the direct-style parser effectively.

6.1.1 *Shifting Statically.* The calls to `ds-parse-bar` in Figure 2 require an application of The Trick: if the symbol argument `symbol` is dynamic, then the result of the function is dynamic as well. All symbol arguments are dynamic. However, for each call to `ds-parse-bar`, the set of symbols which may occur is finite and static. A naive approach would take all grammar symbols as that finite set. However, it is possible to restrict the set further. The first call in `ds-parse` corresponds to a shift on a terminal symbol. With $nextterm(\mathsf{q})$ as the static finite set and with the call to `ds-parse-bar` as context C, the first call has a static value for the `symbol` parameter.

The other calls pass a nonterminal as the symbol argument. The remaining call in `ds-parse` passes `lhs` from the unique item with an empty right-hand side, so this argument is already static. To the recursive call in `ds-parse-bar` we have to apply The Trick once more: in analogy to *nextterm*, the function *nextnonterm* maps a state q to the set with the left-hand side nonterminals of items of the form $\mathsf{A} \to \cdot\alpha$:

$$nextnonterm(\mathsf{q}) := \{\mathsf{A} \mid \mathsf{A} \to \cdot\alpha \in closure(\mathsf{q})\}$$

```
 (define (ds-parse grammar k compute-closure state input)
   (let ((closure (compute-closure state grammar k)))

     (define (reduce)
       (cond
        ((find-lookahead-item (accept closure) k input)
         => (lambda (item)
              (let ((rhs-length (length (item-rhs item)))
                    (lhs (item-lhs item)))
                (if (zero? rhs-length)
                    (ds-parse-bar grammar k compute-closure
                                  closure lhs input)
                    (values lhs rhs-length input)))))
        (else (error "parse error"))))

     (cond
→     ((stream-empty? input) (reduce))
→     ((maybe-the-member (car (stream-car input))
                         (next-terminals closure grammar))
       => (lambda (symbol)
            (ds-parse-bar grammar k compute-closure
                          closure symbol (stream-cdr input))))
      (else (reduce)))))

 (define (ds-parse-bar grammar k compute-closure closure symbol input)
   (let ((the-next-nonterminals (next-nonterminals closure grammar)))
     (call-with-values
      (lambda ()
        (ds-parse grammar k compute-closure
                  (goto closure symbol) input))
      (lambda (lhs dot input)
        (cond
→        ((null? the-next-nonterminals)
→         (values lhs (- dot 1) input))
         ((> dot 1)
          (values lhs (- dot 1) input))
         ((and (initial? closure grammar)
               (equal? (grammar-start grammar) lhs))
          (if (stream-empty? input)
              'accept
              (error "parse error")))
         (else
          (ds-parse-bar grammar k compute-closure
                        closure
→                       (the-member lhs the-next-nonterminals)
                        input)))))))
```

Fig. 8. Direct-style LR(k) parser after binding-time improvements.

```
(define (lookahead-matches? k lookahead input)
  (let loop ((k k) (lookahead lookahead) (input input))
    (cond
     ((zero? k))
     ((null? lookahead) (stream-empty? input))
     ((stream-empty? input) #f)
     ((equal? (car lookahead) (car (stream-car input)))
      (loop (- k 1) (cdr lookahead) (stream-cdr input)))
     (else #f))))

(define (find-lookahead-item item-set k input)
  (let loop ((item-set item-set))
    (if (null? item-set)
        #f
        (let ((item (car item-set)))
          (if (lookahead-matches? k (item-lookahead item) input)
              item
              (loop (cdr item-set)))))))
```

Fig. 9. Checking for lookahead.

The Trick is applicable in exactly the same way as before. By construction of the *goto* function, the left-hand side, `lhs`, of the item returned by `ds-parse` must be a member of *nextnonterm*(`q`). Hence, it is possible to use `the-member` instead of `maybe-the-member` in this case.

Figure 8 shows the parser after the improvements. The listing marks binding-time-improving changes with $\rightarrow$. The reordering of the main conditional requires factoring out the `reduce` action into a procedure; the code itself does not change, however. Moreover, the implementation passes `closure` directly to `ds-parse-bar` instead of recomputing it there from `state`; this speeds up parser generation. The improved parser contains another subtle change to separate binding times: `ds-parse` has to restructure the test for reduction which results in the two calls to the function `reduce`. This is to separate the dynamic test for the end of the input from the static test for the next nonterminal. `And`ing them together as in the original version would make the next nonterminal dynamic.

6.1.2 *Checking for Lookahead.* The other place where The Trick applies is the implementation of `find-lookahead-item`. The loop in the naive implementation is implicit; making it visible to the specializer enables new optimizations, for example, loop unrolling. Figure 9 shows an implementation. Section 8 describes more efficient lookahead-checking strategies which avoid repeated deconstruction of the input stream.

6.2 Generating Code

Binding-time analysis, applied to this parser for a static grammar and static lookahead size, classifies all computations concerned with computing lookahead sets, items, closures, and actions as static. Only direct dependencies on the input and some parse results remain dynamic.

The specializer performs the equivalent of the standard sets-of-items construction on-the-fly using its memoization point mechanism: for each state, specialization generates one version of `ds-parse` and one of `ds-parse-bar`. It is beneficial to cater to the fact that the parser represents states by item lists, since Scheme lacks a native set data type: a set typically has several representations as a list. Therefore, the parser normalizes the states by sorting the items to avoid redundant procedures in the specialized parser.

Furthermore, parser generation performs an optimization on parse tables which is wellknown in the compiler community [Chapman 1987, Chap. 6.3]. Parse tables can be simplified by merging reduce actions with the immediately preceding shift action to a shift/reduce action if no conflicts arise. This transformation happens automatically.

Figure 10 shows specialized code for both parsing procedures. It illustrates nicely how uses of The Trick result in cascaded tests in the specialized program (cf. the tests `(equal? lhs ...)` and `(equal?` *n* `mlet...)`). The specialized code exhibits only two notable differences to hand-optimized code: it contains a few superfluous `let`s, which the simplest of compiler optimizers will inline, and the code accesses the head of the input several times, e.g., `mlet-96`, `mlet-131`, `mlet-173`, and `mlet-187` contain the same value. This duplication arises strictly from the behavior of the original `find-lookahead-item` procedure which also performs the access once for every lookahead check. It would be easy enough to avoid the duplication here, but, as it turns out, a more efficient lookahead procedure which is described in Section 8 takes care of this problem as a side effect.

6.3 Improving Further

Two further optimizations make the generated parsers more efficient.

6.3.1 *Improving The Trick.* Most uses of The Trick in the literature employ the simple linear-search-based implementation shown in Section 5.3.1. However, if there is a total order `<` for the elements of the static set, binary search is possible. If terminals and nonterminals are simply numbers, the Trick loop in `maybe-the-member` changes to the variant shown in Figure 11. A variant of `the-member` results by adding a special `cond` branch for one-element lists.

The call `(sort-list set <)` sorts `set` in ascending order, and `(take middle sorted)` returns the initial segment of length `middle` of list `sorted`.

Whether using binary search actually gives a speed improvement depends highly on the implementation strategies for conditionals used by the underlying Scheme compiler as well as on the grammar at hand. States with a large number of dispatches are rare.

6.3.2 *Using Impure Features.* The major bottleneck in the direct-style implementation results from the necessity to return multiple values from a function—at least the left-hand side of the production currently being reduced, the remaining number of procedure calls still to be returned through, and the remaining input. So far, the return values differ only in their second component. Also, only one of them is live at any given time during the execution. Therefore, it is safe to replace its components by global variables. A new version of the parser uses global variables for the left-hand-side nonterminal `lhs` and the remaining input `input` and pass

```
(define (ds-parse-bar-50 clone-1)
  (let* ((mlet-4 (lambda () (ds-parse-15 clone-1)))
         (mlet-13
           (lambda (lhs_1-5 dot_1-6 input_6-7)
             (cond ((> dot_1-6 1)
                    (values lhs_1-5 (- dot_1-6 1) input_6-7))
                   ((equal? 0 lhs_1-5)
                    (if (stream-empty? input_6-7)
                      'accept
                      (error "parse error")))
                   ((equal? lhs_1-5 3) (ds-parse-bar-46 input_6-7))
                   ((equal? lhs_1-5 2) (ds-parse-bar-47 input_6-7))
                   (else (ds-parse-bar-48 input_6-7))))))
    (call-with-values mlet-4 mlet-13)))
(define (ds-parse-7 input-1)
  (if (stream-empty? input-1)
    (let* ((mlet-34 (stream-car input-1))
           (mlet-33 (car mlet-34)))
      (if (equal? 5 mlet-33)
        (values 2 1 input-1)
        (let* ((mlet-76 (stream-car input-1))
               (mlet-75 (car mlet-76)))
          (if (equal? 6 mlet-75)
            (values 2 1 input-1)
            (let* ((mlet-90 (stream-car input-1))
                   (mlet-89 (car mlet-90)))
              (if (equal? 10 mlet-89)
                (values 2 1 input-1)
                (error "parse error")))))))
    (let* ((mlet-96 (stream-car input-1))
           (mlet-95 (car mlet-96)))
      (cond ((equal? mlet-95 8)
             (ds-parse-bar-8 (stream-cdr input-1)))
            ((equal? mlet-95 7)
             (ds-parse-bar-16 (stream-cdr input-1)))
            (else
             (let* ((mlet-131 (stream-car input-1))
                    (mlet-130 (car mlet-131)))
               (if (equal? 5 mlet-130)
                 (values 2 1 input-1)
                 (let* ((mlet-173 (stream-car input-1))
                        (mlet-172 (car mlet-173)))
                   (if (equal? 6 mlet-172)
                     (values 2 1 input-1)
                     (let* ((mlet-187 (stream-car input-1))
                            (mlet-186 (car mlet-187)))
                       (if (equal? 10 mlet-186)
                         (values 2 1 input-1)
                         (error "parse error")))))))))))
```

Fig. 10. Specialized functional parser.

```
(define (maybe-the-member element set)
  (let loop ((sorted (sort-list set <)) (size (length set)))
    (cond
     ((null? sorted) #f)
     ((>= size 3)
      (let* ((middle (- (quotient (+ size 1) 2) 1))
             (static-element (list-ref sorted middle)))
        (cond
         ((= static-element element)
          static-element)
         ((< element static-element)
          (loop (take middle sorted) middle))
         (else
          (loop (list-tail sorted (+ middle 1))
                (- size (+ middle 1)))))))
     ((= element (car sorted))
      (car sorted))
     (else
      (loop (cdr sorted) (- size 1))))))
```

Fig. 11. The Trick using binary search.

```
 (define *lhs* #f)
 (define *input* #f)

 (define (ds-parse grammar k compute-closure state input)
   ...
                 (ds-parse-bar grammar k compute-closure closure lhs input)
→                (begin (set! *lhs* lhs)
→                       (set! *input* input)
→                       rhs-length)))))
   ...

 (define (ds-parse-bar grammar k compute-closure closure symbol input)
→ (let ((dot (ds-parse grammar k compute-closure
→                (goto closure symbol) input))
→       (lhs *lhs*)
→       (input *input*))
   ...
```

Fig. 12. Direct-style parser using imperative features.

the position of the dot `dot` within the right-hand side of the item as result. Figure 12 shows the changes to the implementation in Figure 2—the modified lines are indicated with $\rightharpoonup$. The resulting speedup of the specialized parsers is impressive.

7. GENERATING EFFICIENT CONTINUATION-BASED LR PARSERS

Just as with the parser in Section 6.1, a few changes are necessary to the naive implementation of the continuation-based parser to make it specialize well. Sometimes, transformation to continuation-passing style makes programs specialize better [Consel and Danvy 1991]. However, our use of continuations in the parser is not motivated by binding-time considerations; rather it is intrinsic in the formulation of the parsing algorithm.

7.1 Termination

The use of continuations in specialization is often problematic, since it can lead to nonterminating specialization. Therefore, we have performed specialization experiments with the intermediate versions of the transformation from the direct-style formulation to the continuation-based one. Two of them have termination problems.

—The version after transformation to continuation-passing style (Figure 3) leads to nonterminating specialization. The binding-time analysis discovers that the continuation is static, and therefore specialization continues to generate new specialized versions of [q] for ever-growing continuation arguments.

—The version after closure conversion (Figure 4) exhibits essentially the same problem. The list d(q,c) is static and leads to the same phenomenon.

—In the final version, pruning the list according to *nactive*(q) is the essential step to ensuring terminating specialization. Section 7.3 exhibits a further improvement to avoid code blow-up.

7.2 Binding-Time Improvements

The Trick applies in the same situations as in the direct-style parser. However, the compact specification of the continuation-based parser has only one place where shifting occurs (the continuation), whereas in the direct-style version separate function calls are responsible for shifting on terminals and nonterminals, respectively. Hence, we only know that the `symbol` argument to `c0` is a member of *nextterm*(`state`) $\cup$ *nextnonterm*(`state`). Applying The Trick naively would specialize to one loop that tests `symbol` against both terminals and nonterminals.

However, the parser calls `c0` from two different call sites. The first site only ever passes nonterminals, the second only terminals. Thus, splitting `c0`—into one version for nonterminals and another for terminals—and applying The Trick in the same way as with the direct-style parser yields the desired results from specialization. Curiously, this amounts to undoing the clever steps leading from the result of the transformation in Figure 5 to Sperber's formulation in Figure 6.

7.3 Removing the List of Continuations

Using standard Scheme lists for storing the continuations introduces a performance problem with most partial evaluators [Bondorf 1993]: these partial evaluators treat `cons`, `car`, and `cdr` as ordinary primitive operations, rather than as data constructors and selectors. Traditional monovariant binding-time analyses [Bondorf 1991] expect the arguments to a primitive to be static data of base type if the specializer is to reduce the primitive. Since our parser passes a procedure to the primitive `cons`, the binding-time analysis detects a type clash and defers the expected error to run time by marking the `cons` as dynamic. Therefore, the specialized parsers presently construct and pass lists of continuations. This is grossly inefficient, as the structure of the list is static.

```
→(define-data c-list
→ (c-cons c-car c-cdr)
→ (c-nil))

 (define (cps-parse grammar k compute-closure state continuations input)
   (let* ((closure (compute-closure state grammar k))
          (accept-items (accept closure))
          (the-next-nonterminals (next-nonterminals closure grammar)))

→    (define (shift symbol input)
       (let ((next-state (goto closure symbol)))
         (cps-parse grammar k compute-closure
                    next-state
→                   (c-cons (and (not (null? the-next-nonterminals))
                                 shift-nonterminal)
→                           (c-take (- (nactive next-state) 1)
                                    continuations))
                    input)))

     (define (shift-nonterminal nonterminal input)
→      (if (and (initial? state grammar)
→               (equal? (grammar-start grammar) nonterminal))
→          (if (stream-empty? input)
               'accept
               'error)
           (shift
→           (the-member nonterminal the-next-nonterminals)
            input)))

     (define (reduce item)
→      ((c-list-ref (c-cons (and (not (null? the-next-nonterminals))
                                 shift-nonterminal)
                            continuations)
                    (length (item-rhs item)))
        (item-lhs item) input))

     (cond
→     ((stream-empty? input)
→      (cond
→       ((find-eoi-lookahead-item accept-items) => reduce)
→       (else 'error)))
→     ((maybe-the-member (car (stream-car input))
                         (next-terminals closure grammar))
       => (lambda (symbol)
            (shift symbol (stream-cdr input))))
      ((find-lookahead-item accept-items k input) => reduce)
      (else 'error))))
```

Fig. 13. Continuation-based parser after binding-time improvements.

In the specification, the parser states simply pass the continuations as separate arguments. To achieve the same effect in the specialized parsers, it is necessary to convey the structure of the list of continuations to the binding-time analysis and to the specializer. Most partial evaluators support *partially static data structures* for just that purpose: they allow the construction of aggregates which contain both static and dynamic components. In our case, the spine of the list is known. Hence, declaring a partially static list datatype to the partial evaluator and replacing `cons`, `car`, and `cdr` by their counterparts achieves the desired purpose. The binding-time analysis keeps the spine of the list static. The static reducer performs "arity raising" and creates a separate parameter for each list element. Splitting up the list of continuations is crucial for the performance of the specialized parsers.

However, with this change, the binding-time analysis actually propagates too much information. It maintains that the functions in the list are also static which prompts the specializer to generate separate versions of each state for each possible constellation of continuations that may be its parameters, resulting in code blow-up. Therefore, we have to force the binding-time analysis to regard the elements of the `continuations` list itself as dynamic. This is typically achieved by a special operator provided by the partial evaluator at hand which forces its argument to become dynamic. Applying the operator to the occurrences of `c0` has the desired effect.

Our final implementation achieves this effect by a different means. If the parser is in a state without an item of the form $A \rightarrow \cdot\alpha$ then the parser will never shift on a nonterminal in this state. Therefore, in such a state, the parser puts a dummy value `#f` on the list of continuations and never bothers to construct the actual continuation. Therefore, an element of the `continuations` list might either be a function (an actual continuation) or a boolean `#f`. The binding-time analysis detects the apparent type clash and makes all the list elements dynamic to avoid an error at specialization time. The corresponding test in the code is `(not (null? the-next-nonterminals))`.

Figure 13 shows the parser with the binding-time improvements applied. Changes are again marked with $\rightarrow$ symbols. Just as with the direct-style parser, the improved version factors out the reduce action into a separate procedure `reduce`.

7.4 Generating Code

The results of specializing the continuation-based parser are similarly satisfying as those from specializing the direct-style version. Figure 14 shows a sample. In the sample, the `shift-nonterminal` function previously internal to `cps-parse` has moved to the top level, as the PGG system performs Lambda lifting in the frontend [Johnsson 1985]; this is purely a specific property of PGG. As the code was generated using the same version of `find-lookahead-item`, it shows the same duplication of the calls to `stream-car` and `car`—to be remedied in the next section.

8. EFFICIENT CHECKING FOR LOOKAHEAD

The specialized parsers should check for lookahead efficiently. Specifically, they should touch as few input characters as possible, and they should perform as few operations as possible to determine the next state of the parser. As seen in Sec-

```
(define (cps-parse-19 mlet-1 mlet-2 clone-3 clone-4)
  (if (stream-empty? clone-4)
    (mlet-1 3 clone-4)
    (let* ((mlet-7 (stream-car clone-4))
           (mlet-6 (car mlet-7)))
      (if (equal? 5 mlet-210)
        (mlet-1 3 clone-4)
        (let* ((mlet-481 (stream-car clone-4))
               (mlet-480 (car mlet-481)))
          (if (equal? 6 mlet-480)
            (mlet-1 3 clone-4)
            (let* ((mlet-571 (stream-car clone-4))
                   (mlet-570 (car mlet-571)))
              (if (equal? 7 mlet-570)
                (mlet-1 3 clone-4)
                (let* ((mlet-601 (stream-car clone-4))
                       (mlet-600 (car mlet-601)))
                  (if (equal? 8 mlet-600)
                    (mlet-1 3 clone-4)
                    (let* ((mlet-611 (stream-car clone-4))
                           (mlet-610 (car mlet-611)))
                      (if (equal? 10 mlet-610)
                        (mlet-1 3 clone-4)
                        'error)))))))))))

(define (shift-nonterminal-3 nonterminal-2 input-1)
  (cond ((equal? 0 nonterminal-2)
         (if (stream-empty? input-1) 'accept 'error))
        ((equal? nonterminal-2 3)
         (let ((mlet-6
                (lambda (nonterminal_3-3 input_9-4)
                  (shift-nonterminal-3 nonterminal_3-3 input_9-4))))
           (cps-parse-4 mlet-6 input-1)))
        ((equal? nonterminal-2 2)
         (let ((mlet-11
                (lambda (nonterminal_3-8 input_9-9)
                  (shift-nonterminal-3 nonterminal_3-8 input_9-9))))
           (cps-parse-10 mlet-11 input-1)))
        (else
         (let ((mlet-16
                (lambda (nonterminal_3-13 input_9-14)
                  (shift-nonterminal-3 nonterminal_3-13 input_9-14))))
           (cps-parse-23 mlet-16 input-1)))))
```

Fig. 14. Specialized continuation-based parser.

```
(define (find-lookahead-item item-set k input)
  (let loop ((lookaheads+items (map (lambda (item)
                                      (cons (item-lookahead item)
                                            item))
                                    item-set))
             (remaining k)
             (input input))
    (cond
     ((null? lookaheads+items)
      #f)
     ((zero? remaining)
      (cdar lookaheads+items))
     ((and (not (= k remaining))
           (stream-empty? input))
      (let ((empties
             (filter (lambda (lookahead+item)
                       (null? (car lookahead+item)))
                     lookaheads+items)))
        (if (null? empties)
            #f
            (cdar empties))))
     (else
      (loop (filter-lookaheads+items lookaheads+items
                                     (car (stream-car input)))
            (- remaining 1)
            (stream-cdr input))))))

(define (filter-lookaheads+items lookaheads+items terminal)
  (let* ((non-empties (filter (lambda (lookahead+item)
                                (not (null? (car lookahead+item))))
                              lookaheads+items))
         (one-lookaheads (map (lambda (lookahead+item)
                                (car (car lookahead+item)))
                              non-empties))
         (static-terminal (maybe-the-member terminal
                                            one-lookaheads))
         (matches
          (filter
           (lambda (lookahead+item)
             (equal? static-terminal (caar lookahead+item)))
           non-empties)))

    (map (lambda (lookahead+item)
           (cons (cdr (car lookahead+item))
                 (cdr lookahead+item)))
         matches)))
```

Fig. 15. Efficient checking for lookahead with tries.

tions 6.2 and 7.4, the generated code still contains redundant accesses to the head of the input. This problem gets worse with $k > 1$.

Figure 15 displays the implementation of a variant of The Trick. The variant of `find-lookahead-item` successively narrows down the set of items that matches the current lookahead. For this purpose, `find-lookahead-item` pairs up the remaining lookaheads with their respective items and uses `filter-lookaheads+items` to first weed out the items which do not match the next single lookahead, and subsequently remove that one lookahead from the lookahead-item pairs. The `filter` function which the code employs takes a predicate and a list as arguments. It returns a list of exactly those elements that match the predicate. The new version effectively encodes tries and therefore neither performs redundant checks nor extraneous accesses to the input.

9. ADDITIONAL FEATURES

The presented parsers so far are LR(k) *recognizers*. A few modifications and additions are necessary to make them useful for applications: SLR and LALR parsing, attribute evaluation, and an error recovery mechanism. Partial evaluation-based parser generation is amenable to these additions.

9.1 SLR and LALR

Pure LR parsers for realistic languages often lead to prohibitively big state automatons [Chapman 1987], and thus to impractically big parsers. Fortunately, most realistic formal languages are already amenable to treatment by SLR or LALR parsers which introduce lookahead into essentially LR(0) parsers.

The SLR(k) parser corresponding to an LR(0) parser [DeRemer 1971] with states $q_0^{(0)}, \ldots, q_n^{(0)}$ has states $q_0, \ldots, q_n$. In contrast to the LR(k) parser, the SLR(k) automaton has the following states:

$$q_i := \{A \to \alpha \cdot \beta \ (\rho) \mid A \to \alpha \cdot \beta \in q_i^{(0)}, \rho \in \mathit{follow}_k(A)\}$$

In the definition, $\mathit{follow}_k(A)$ is the set of all terminal sequences of length $\leq k$ that may follow A in a sentential form.

$$\mathit{follow}_k(A) := \{\xi \mid S \overset{*}{\Rightarrow} \beta A \gamma \ \wedge \ \xi \in \mathit{first}_k(\gamma)\}$$

Analogously, the predict items are the same as in the LR(0) case, only with added lookahead:

$$\mathit{predict}(q_i) := \{A \to \alpha \cdot \beta \ (\rho) \mid A \to \alpha \cdot \beta \in \mathit{predict}^{(0)}(q_i^{(0)}), \rho \in \mathit{follow}_k(A)\}$$

The state transition *goto* is also just a variant the LR(0) case here called $\mathit{goto}^{(0)}$:

$$\mathit{goto}(q_i, X) := q_j \text{ for } q_j^{(0)} = \mathit{goto}^{(0)}(q_i^{(0)}, X)$$

It is immediately obvious how to modify an LR(0) parser into an SLR(k) parser—the main parsing function merely has to replace the current state by one decorated by lookahead as described above. The effects of using SLR(k) instead of LR(k) are as expected: generation time and size decrease, often dramatically for realistic grammars (see Table I in the next section).

The LALR method uses a more precise method of computing the lookahead, but also works by decorating an LR(0) parser [DeRemer 1969]. Thus, the same

```
$[q](\xi, c_1, \dots, c_{nactive(q)}, s_1, \dots, s_{nactive(q)}) :=$
  letrec
    $c_0(X, s_0, \xi) = [goto(q, X)](\xi, c_0, c_1, \dots, c_n, s_0, s_1, \dots, s_n)$
                                                where $n = nactive(goto(q, X)) - 1$
  in
    $A \to \alpha \cdot\ (\rho) \in closure(q) \wedge \xi_{|k} = \rho$  $\triangleright$  $c_{|\alpha|}(A, f_{A\to\alpha}(s_{|\alpha|}, \dots, s_1), \xi)$
  | $\xi = x\xi' \wedge x \in nextterm(q)$                  $\triangleright$  $c_0(x, \Box, \xi')$
```

Fig. 16. Continuation-based LR(k) parser with attribute evaluation.

methodology as with the SLR case is applicable, merely replacing *follow*$_k$ with the (more involved) LALR lookahead function.

Unfortunately, all efficient methods of computing LALR lookahead sets require access to the entire LR(0) automaton in advance [DeRemer and Pennello 1982; Park et al. 1985; Ives 1986; Park and Choe 1987; Ives 1987a; 1987b]. Since our general parser does not have access to a representation of the automaton, introducing the LALR method would require computing one and operating on that, instead of relying on the partial evaluator to implicitly generate it. The feasibility of the approach is not impaired by this, but explicitly computing the automaton somewhat breaks its spirit and elegance.

9.2 Attribute Grammars

The pure recognizers used so far are of little use in practice. To take advantage of the results of the parsing process, the parser must associate a semantics with the input string. The most common way to do this is the *attribute grammar* mechanism [Knuth 1968; 1971]. Many popular parser generators such as Yacc [Johnson 1975] restrict themselves to *S-attributed grammars* where all attributes can be evaluated "on-the-fly" during parsing. This is entirely sufficient for parsers whose principal job is to compute an abstract representation of the syntax tree. Parsers can propagate additional values such as symbol tables via global variables. However, more general attribute evaluation strategies are also possible, and the parsers presented here are in fact especially amenable to them [Sperber 1994]. Moreover, higher-order functions can simulate more general attribution schemes within the framework of S-attributed grammars [Chirica and Martin 1979].

An S-attributed grammar is a context-free grammar where each nonterminal A is associated with a *synthesized attribute* s_A, while each production is augmented with an *attribution*. An attribution of a production $A \to X_1 \dots X_n$ is a function $f_{A\to X_1 \dots X_n}$ which maps the synthesized attributes of the X_i, $s_1 \dots s_n$ to the synthesized attribute of A via

$$s_A = f_{A\to X_1 \dots X_n}(s_1, \dots, s_n).$$

A straightforward extension of the direct-style parsers and the continuation-based parsers performs attribute evaluation for S-attributed grammars on-the-fly—while parsing the input. Attribute evaluation in the direct-style parser requires an explicit attribute stack. The continuation-based parser handles attributes in the same way as the list of continuations, as a partially static list that contains the attributes

```
  (define(cps-parse grammar k compute-closure state
                    continuations attribute-values input)
   (let* ((closure (compute-closure state grammar k))
          (accept-items (accept closure))
          (the-next-nonterminals (next-nonterminals closure grammar)))

    (define (shift symbol attribute-value input)
      (let* ((next-state (goto closure symbol))
             (keep (- (nactive next-state) 1)))
        (cps-parse grammar k compute-closure
           next-state
           (c-cons (and (not (null? the-next-nonterminals)) shift-nonterminal)
                   (c-take keep continuations))
→          (c-cons attribute-value (c-take keep attribute-values))
           input)))

    (define (shift-nonterminal nonterminal attribute-value input)
      (if (and (initial? state grammar)
               (equal? (grammar-start grammar) nonterminal))
          (if (stream-empty? input)
→             attribute-value
              (error "parse error"))
          (shift (the-member nonterminal the-next-nonterminals)
                 attribute-value input)))

    (define (reduce item)
      (let* ((rhs-length (length (item-rhs item)))
             (attribution (production-attribution (item-production item)))
→            (attribute-value
→             (apply-attribution
→              attribution
→              (c-list->list
→               (c-reverse
→                (c-take rhs-length attribute-values))))))
        ((c-list-ref (c-cons (and (not (null? the-next-nonterminals))
                                  shift-nonterminal)
                             continuations)
                     rhs-length)
         (item-lhs item) attribute-value input)))

    (cond ((stream-empty? input)
              (cond ((find-eoi-lookahead-item accept-items) => reduce)
                    (else (error "parse error"))))
     ((maybe-the-member (car (stream-car input))
                        (next-terminals closure grammar))
      => (lambda (symbol)
           (shift symbol (cdr (stream-car input)) (stream-cdr input))))
     ((find-lookahead-item accept-items k input) => reduce)
     (else (error "parse error")))))
```

Fig. 17. Implementation of continuation-based parser with attribute evaluation.

$$
\begin{array}{lll}
[q](\xi, s) & := & x\xi' \leftarrow \xi \\
 & & \quad \rhd \ \overline{[q]}(x, \xi', \Box, s) \\
 & | & B \rightarrow \cdot\ (\rho) \in \mathit{predict}(q) \wedge \xi_{|k} = \rho \\
 & & \quad \rhd \ \overline{[q]}(B, \xi, f_{B \rightarrow \varepsilon}(), s) \\
 & | & A \rightarrow \alpha \cdot\ (\rho) \in q \wedge \xi_{|k} = \rho \\
 & & \quad \rhd \ (A \rightarrow \alpha\cdot, \xi, f_{A \rightarrow \alpha}(s_{|\alpha|}, \ldots, s_1))
\end{array}
$$

$$
\begin{array}{lll}
\overline{[q]}(X, \xi, s_0, s) & := & A \rightarrow \alpha \cdot X\gamma \in q \wedge (A \rightarrow \alpha X \cdot \gamma, \tau, s_0') \leftarrow [\mathit{goto}(q, X)](\xi, s_0 : s) \\
 & & \quad \rhd \ (A \rightarrow \alpha \cdot X\gamma, \tau, s_0') \\
 & | & C \rightarrow \cdot X\delta \in \mathit{predict}(q) \wedge (C \rightarrow X \cdot \delta, \tau, s_0') \leftarrow [\mathit{goto}(q, X)](\xi, s_0 : s) \\
 & & \quad \rhd \ \overline{[q]}(C, \tau, s_0', s)
\end{array}
$$

Fig. 18. Direct-style LR(k) parser with attribute evaluation.

for the currently active symbols. It is simpler than the direct-style version and therefore presented first.

The continuation-based parser adheres to the specification in Figure 16. The attribute value domain is assumed to contain a value $\Box$ which the parser assigns to terminals. In the implementation, the parser accepts sequences of token/attribute pairs, so that terminals may also carry attribute information. Implementation—shown in Figure 17—is straightforward.

The specification of the direct-style parser with on-the-fly attribute evaluation shown in Figure 18 is somewhat more complicated. Now, $[q]$ accepts an input string and an attribute stack. As in a traditional LR parser, the attribute stack contains the values of the attributes of all symbols currently on the parse stack. The function returns a triple consisting of an LR item, the remaining input, and the value of the synthetic attribute of the left-hand-side nonterminal of the item. (The attribute value is generated when reduction of an item $A \rightarrow \alpha\cdot$ is initiated.) $\overline{[q]}$ is similarly extended to deal with the attribute stack and the attribute value of the shifted symbol.

The attribute stack is implemented as a list s with elements $s_1 s_2 \ldots s_n$. The infix operator : puts an element in front of the list (implementing the push operation). The implementation never removes elements explicitly, but returning from a function invocation of $\overline{[q]}$ "forgets" the first element of s. In principle, the algorithm could crop the lists just like the algorithm in Figure 16, but we left them as is to keep the analogy with the attribute stack in a traditional implementation.

In order to perform attribute evaluation the attribution of each production must be transferred to the generated parser. One way to achieve attribute evaluation is to interpret the attributions:

```
(interpret (cons (attribution->definition attribution)
                 program)
           (take rhs-length attribute-stack))
```

Unfortunately, this approach has several drawbacks. First, the interpreter must be written, which is straightforward, but rather tedious, especially if full Scheme is to be supported. Second, parser generation becomes grossly inefficient because the partial evaluator must specialize the interpreter which actually runs the code of the attribution—at least one superfluous layer of interpretation.

A much better choice consists in using `eval` to transform the text of the attribution into the actual code:

```
(apply (eval attribution (interaction-environment))
       (take rhs-length attribute-stack))
```

Since `attribution` will be static, the specializer could remove `eval` from the code and simply paste in the static value of `attribution`, considered as code. For example, for the attribution `(lambda ($1 $2 $3) (* $1 $3))` the specializer should generate the following code:

```
(let ((var-9031 ((lambda ($1 $2 $3)
                    (* $1 $3))
                  (car var-9030)
                  (car var-9029)
                  (car clone-9018))))
  ...)))
```

The partial evaluator used for the implementation of the parsers handles `eval` and `apply` correctly [Thiemann 1996b] which results in a clear, concise, and efficient solution to the problem.

9.3 Error Recovery

Realistic applications of parsers require sensible handling of parsing errors. Specifically, the parser should, on encountering a parsing error, issue an error message and resume parsing in some way, repairing the error if possible. The literature abounds with theoretical treatments of recovery techniques applicable to LR parsing [Chapman 1987; Sippu and Soisalon-Soininen 1990] using a wide variety of methods. Most of these methods are *phrase-level recovery techniques* that work by transforming an incorrect input into a correct one by deleting terminals from the input and inserting generated ones. Among the many phrase-level recovery techniques, few have actually been used in production LR parser generators.

To demonstrate the feasibility of effective error recovery in our approach, we have implemented two realistic algorithms: the grammar-assisted method of Yacc [Johnson 1975] based on error productions and the fully automatic technique of Röhrich [1980] as implemented in the Lalr parser generator [Grosch 1990].

Since effective error recovery requires fine-grained control over the operation of the parser, the continuation-based parser is considerably more amenable to the implementation of these techniques—hence, the treatment of error recovery is entirely in the context of the continuation-based parsers.

9.3.1 *Yacc-Style Error Recovery.* Yacc provides a special "error terminal" as a means for the user to specify recovery annotations. A typical example is the following grammar for arithmetic expressions E (terminals such as `number`, `+`, etc., are in typewriter font):

$$\begin{aligned}
\mathsf{E} &\rightarrow \mathsf{T} \mid \mathit{error} \mid \mathsf{T}\texttt{+}\mathsf{E} \mid \mathsf{T}\texttt{-}\mathsf{E}\\
\mathsf{T} &\rightarrow \mathsf{P} \mid \mathsf{P}\texttt{*}\mathsf{T} \mid \mathsf{P}\texttt{/}\mathsf{T}\\
\mathsf{P} &\rightarrow \texttt{number} \mid \texttt{(}\mathsf{E}\texttt{)} \mid \texttt{(}\mathit{error}\texttt{)}
\end{aligned}$$

Whenever the parser encounters a parsing error, it pops symbols from the parse stack until it reaches a state where it can shift on the error terminal. There, the parser shifts and then skips input terminals until the next input symbol is acceptable to the parser again. In the example, the parser, when encountering an error in a parenthesized expression, will skip until the next closing parenthesis.

To keep the error messages from avalanching, the parser needs to keep track of the number of terminals that have been shifted since the last error; if the last error happened very recently, chances are the new error has actually been effected by the error recovery. In that case, the parser should skip at least one input terminal (to guarantee termination), and refrain from issuing another error message.

This method is fairly crude, but has proven effective. It also has the advantage over fully automatic methods that it provides the grammar author with the ability to tailor specific error messages to the context of a given error and specify sensible attribute evaluation rules.

In the continuation-based parser, the Yacc method is fairly straightforward to implement. In addition to the usual continuations to perform reductions, we supply an *error continuation* `handle-error` which brings the parser back immediately into the last state to shift on the error terminal [Leermakers 1993]. In addition, another parameter `error-status` keeps track of the number of terminals that the parser still needs to shift until it can resume issuing error messages; in our case that number is three. Thus, all calls to `cps-parse` pass the `handle-error` and `error-status` parameters like this:

```
(cps-parse
  ...
  (if (handles-error? closure grammar)
      handle-error-here
      handle-error)
  ...)
```

(`Handle-error-here` needs to be made dynamic to prevent nonterminating specialization.)

Whenever the parser shifts on a terminal, it adjusts `error-status` to

```
  (if (zero? error-status) error-status (- error-status 1))
```

Figure 19 shows the relevant `handle-error-here` function.

Thus, Yacc-style error recovery fits our parser model very naturally, and calling the error continuation directly is certainly more efficient than scanning the stack explicitly, as Yacc needs to do. Additionally, we have the option to keep the `error-status` count static. This results in an increase in code size, but also in a small gain in parser speed.

9.3.2 *Fully Automatic Backtrack-Free Error Recovery.* In contrast to the simple model of Yacc, the sophisticated method of Röhrich [1980] allows for (almost) fully automatic error recovery. The price is added complexity and lower performance. We only sketch the method here, since the details of the algorithm are rather involved. Röhrich's algorithm constructs a *continuation automaton* from the parsing automaton which in turn constructs a valid continuation of the input from a valid prefix (a prefix of a correct input). The continuation automaton is merely a special

```
(define (handle-error-here error-status input)
  (let* ((next-state (goto closure (grammar-error grammar)))
         (keep (- (nactive next-state) 1))
         (next-closure (compute-closure next-state grammar k))
         (next-accept-items (accept next-closure))
         (input
          (cond
           ((zero? error-status) input)
           ((stream-empty? input)
            (error "parse error: premature end of input"))
           (else (stream-cdr input)))))

    (define (recover attribute-value input)
      (cps-parse grammar k compute-closure
                 next-state
                 (c-cons (and (not (null? the-next-nonterminals))
                              shift-nonterminal)
                         (c-take keep continuations))
                 (c-cons attribute-value
                         (c-take keep attribute-values))
                 handle-error-here 3
                 input))

    (let loop ((input input))
      (define (reduce-recover item)
        (let* ((rhs-length (length (item-rhs item)))
               (attribution (production-attribution
                             (item-production item)))
               (attribute-value
                (apply-attribution
                 attribution
                 (c-list->list
                  (c-reverse
                   (c-cons #f (c-take (- rhs-length 1)
                                      attribute-values)))))))
          (recover attribute-value input)))
      (cond
       ((stream-empty? input)
        (cond
         ((find-eoi-lookahead-item next-accept-items) => reduce-recover)
         (else (error "parse error: premature end of input"))))
       ((maybe-the-member (car (stream-car input))
                          (next-terminals next-closure grammar))
        => (lambda (symbol)
             (recover (cdr (stream-car input)) input)))
       ((find-lookahead-item next-accept-items k input)
        => reduce-recover)
       (else (loop (stream-cdr input)))))))
```

Fig. 19. Yacc-style error recovery.

```
(begin
  ;; report the error
  (display "Syntax error: expected one of ")
  (write (uniq (append (next-terminals closure grammar)
                       (flatten (items-lookaheads accept-items)))))
  (newline)

  (set! *repairing?* #t)
  (set! *anchors* '())

  ;; construct the set of anchors
  (cps-parse grammar k compute-closure
             state
             continuations attribute-values
             input)

  ;; skip until anchor
  (let ((input
         (let loop ((input input))
           (if (and (not (stream-empty? input))
                    (member (car (stream-car input)) *anchors*))
               input
               (begin
                 (display "Deleting ")
                 (write (car (stream-car input)))
                 (newline)
                 (loop (stream-cdr input)))))))

    (set! *anchors* #f)
    (set! *repairing* #t)

    ;; repair
    (cps-parse grammar k compute-closure
               state
               continuations attribute-values
               input)))
```

Fig. 20. Röhrich-style error handling.

interpretation of a variant of the LR state automaton. In the variant, the states are no longer sets of items but, rather, sorted sequences. This makes the automaton with error recovery larger than the canonical LR automaton.

On encountering an error, the algorithm performs the following steps:

(1) First, it simulates running the parser on a constructed continuation of the current correct prefix until parsing terminates. All terminals of the continuation form the set of *anchors* which are valid restarting points for parsing.

(2) The parser deletes terminals from the input until the next terminal is one of the anchors.

(3) Then, the parser synthesizes a continuation (the same as generated in the first step) up until the anchor from where it can resume parsing.

In a parser that interprets a representation of the LR state table, the simulation of the automaton is straightforward. To simulate running the parser, it merely needs to copy the parsing stack so that it can be restored when actual parsing resumes. In our continuation-based parser, we have essentially two choices:

(1) construct separate functions for actually running the parser and simulating it or

(2) use the same functions for each task and keep track of the difference by a local state.

The first alternative has the disadvantage that partial evaluation would create two (or even three) variants of the parser, one for regular parsing, one for collecting anchors, and one for continuing the input. We have therefore chosen to use the regular parsing functions and keep a global state. One is the `*repairing?*` flag which is true when error recovery is in progress. The other state component is `*anchors*` which contains the current list of anchors as they are being collected.

Figure 20 shows the code that the parser executes on encountering a parsing error. For brevity's sake, it does not perform attribute evaluation. Once the parser is in "repair mode," it checks in each state if it can continue parsing with the current lookahead. If not, the state has an associated continuation terminal which the parser can use instead. In that case, it must also generate an attribute value for the generated terminal which is supplied as part of the source grammar.

10. EXPERIMENTAL RESULTS

With both the direct-style and the continuation-based implementation models, the generated LR parsers compare favorably with those generated by traditionally built parser generators such as Bison [Donnelly and Stallman 1995] as well as those produced by the partial evaluation of a stack-based implementation presented by Mossin [1993].

The two implementation models have different merits; while the direct-style approach leads to more compact parsers, the continuation-based parsers are faster in most cases. We have applied both Mossin's parser generator as well as our continuation-based and direct-style approaches to three different example grammars. To confine the results to parsing proper, the generated parsers do not perform attribute evaluation or error recovery.

Table I shows the sizes of the input grammars and the sizes of the generated parsers. The sizes are given in the number of cells used for the Scheme representations. The column "Mossin" gives numbers for parsers generated from Mossin's general parser; "LR" is for the direct-style parsers; and "CPS-LR" is for continuation-based parsers. In column "SLR" we give the sizes of the specialized direct-style SLR(1) parsers. The example grammars are those used by Mossin: G_1 defines balanced parentheses, G_2 arithmetic expressions, and G_3 the language Mixwell [Jones et al. 1985].

Table II shows the run times of the generated parsers over different grammars and inputs of varying sizes. The measurements were taken on an IBM RS/6000

Table I. Size of Specialized Parsers for $k = 1$ (in cons-cells)

G	Size (G)	Mossin	LR	CPS-LR	SLR
G_1	24	1608	652	1236	491
G_2	48	3751	2070	2999	1197
G_3	123	6181	5870	7294	2700

Table II. Run Times of the Specialized Parsers (timings in ms)

G	Size (input)	Mossin	LR	CPS-LR	LR-Imp	Bison
G_1	2	0.0637	0.0986	0.0480	0.0439	0.0220
	8	0.2030	0.2465	0.1343	0.0747	0.0586
	28	0.6376	0.7474	0.4470	0.1896	0.1785
G_2	3	0.1372	0.1732	0.0840	0.0554	0.0379
	13	0.4213	0.6273	0.3689	0.1254	0.1123
	35	0.9957	1.3755	0.9294	0.2605	0.2600
G_3	9	0.1293	0.3509	0.1211	0.0671	0.0600
	51	0.7699	1.6911	0.7523	0.3012	0.2847
	186	2.6190	5.8680	2.9020	1.0130	0.9470
	983	12.4240	31.0130	13.4630	4.5970	4.6280

model 320 with 24MB of real memory running AIX >3.2.5. We used Bigloo 1.7, a Scheme compiler which generates C code, with maximum optimization. The C code was compiled using the native C compiler, xlc 1.3. The column "LR-Imp" shows the results for an imperative version of the direct-style parser along the lines of Figure 12. The last column shows timings for equivalent LALR(1) parsers in C generated by Bison 1.22 and compiled with maximum optimization. The input for the Bison parser was fed directly from a constant array containing the token codes. Thus, all timings measure purely the parsing time.

The timings indicate that the speed of our functional parsers is within a factor of 2 of (with direct-style parsing) or surpasses (with the continuation-based approach) those generated from the stack-based approach by Mossin [1993]. The imperative direct-style version even surpasses those timings and gets very close to the Bison-generated parsers in speed. These results indicate the practicability of our approach.

Moreover, we have used the continuation-based parser generator to develop the front end of an ANSI C compiler [Kernighan and Ritchie 1988].

Since it is possible to test a given grammar with the general parser, our observation has been that users typically only apply the parser generator toward the end of the development cycle. During development, the general parser allows for almost instant turnaround.

11. RELATED WORK

The pioneering work on functional LR parsing is due to Pennello [1986]. He gives a low-level implementation of a direct-style functional parser. He reports a speedup of 6 to 10 over implementations that utilize a table representation of the parsing automaton. As a matter of fact, the Bison parser generator [Donnelly and Stallman 1995] which we used for comparisons in our benchmarks is much faster than the parser generator implementation Pennello used; it seems unlikely that significant performance improvements are possible over the parser model presented here.

Leermakers [1993] provides an overview of functional parsing. A summary along with a description of the continuation-based approach is given by Sperber [1994]. He uses the continuation-based parsing algorithm to implement sophisticated attribute evaluation. The resulting algorithms are used in the parser generator of the Mørk system for preprocessor generation.

As already mentioned in this article, Mossin [1993] uses Similix [Bondorf 1993] to obtain specialized LR(1) parsers from general parsers. He starts from a stack-based first-order approach which does not specialize well. He transforms the stack data structure into a continuation which pops elements off the stack. The transformation complicates the program considerably and requires intricate binding-time improvements and other optimizations to achieve good specialization.

Dybkjær [1985] as well as Ershov and Ostrovsky [1987] describe previous attempts to specialize general parsers.

Pagan [1991] describes, among other examples for partial computation, the construction of LL and LR parser generators. However, his approach is rather adhoc, and the generation of the parser generators is not automatic but done by hand.

Consel and Danvy [1993] give an overview of partial evaluation. Jones et al. [1993] give a more in-depth study of techniques and applications of the field.

The idea of self-application and its use for semantics-based compilation stems from Futamura [1971] and has been refined by Turchin [1979]. Since then, compiler generation and self-application have been among the main fields of interest for researchers in partial evaluation. This led to the discovery of offline partial evaluation and the construction of practical compiler generators [Jones et al. 1985].

Instead of using a partial evaluator and self-application to create standalone program generators, it is also possible to use a hand-written program-generator generator [Launchbury and Holst 1991] which also boosts efficiency and solves some coding problems as compared to using self-application. We have used such a program-generator generator [Thiemann 1996a] to generate our parser generators.

The Trick seems to be as old as offline partial evaluation [Dybkjær 1985; Gomard and Jones 1991; Bondorf 1993; Jones et al. 1993], as it is necessary for effective self-application. No existing offline partial evaluator automates The Trick. However, there is a formalization that incorporates The Trick into a binding-time analysis for the lambda calculus and defines context propagating rules for static reductions [Danvy et al. 1996].

Swierstra and Duponcheel [1996] define a combinator library for recursive-descent parsers from LL(1) grammars in a lazy functional language based on earlier work by Hutton [1990]. These parsers appear to specialize themselves by exploiting the laziness of the language. There are two differences to our approach, despite the fact that we are concerned with LR parsing. First, their combinators exploit the memoization capabilities of the lazy evaluation. The effects are similar to what specialization can achieve, but Holst and Gomard [1991] have shown that specialization is strictly stronger than lazy evaluation (even in the presence of full laziness), at least in a higher-order programming language. Second, while our parser generators can check whether the grammar satisfies the LR(k) (or SLR (k)) condition, the parser combinators loop when used for a grammar which is not LL(1).

Parser generation is a typical application of offline partial evaluation. Other such applications are compilation, compiler generation, and program transformation. These applications share similar binding-time improvements which lead to good specialization, as well as the necessity of a memoization point mechanism to obtain termination. The common improvement of using partial evaluation for these applications is the removal of interpretive overhead. Consequently, the speedup obtained depends mostly on the proportion of overhead in the original programs. Jones et al. [1993] describe a number of these applications. They are in contrast to applications of offline partial evaluation in, for example, scientific computing. Here, the main benefit of specialization is constant propagation and loop unrolling, just as in an optimizing compiler. Hence, specialization generally obtains mostly constant speedups in these areas.

12. CONCLUSION

We have used partial evaluation to generate fast and compact LR(k) parsers from a general functional parser and a reference grammar. We used two approaches to implement the general parser: one which represents the parsing stack by the procedure call stack, the other using continuations. No sacrifice of generality is necessary to make the programs amenable to good specialization. There is no need to cater to specific optimizations used in parser generators, or to $k = 1$.

The functional approach needs only straightforward and well-known binding-time improvements to generate efficient specialized parsers.

The generated parsers are fast and compact, and they compare favorably with Yacc-generated parsers. Consequently, partial evaluation is a realistic approach to implementing production-quality parser generators.

APPENDIX

The parser generators constructed by transforming the general parsers described in this article to generating extensions is available on the Worldwide Web under `http://www.informatik.uni-freiburg.de/proglang/software/essence/`. Besides parser generators which are ready to run, the distribution also contains the source for various general LR parsers, suitable for specialization with the PGG system. For the more adventurous, the PGG system is also available from

http://www.informatik.uni-freiburg.de/proglang/software/pgg/

ACKNOWLEDGMENTS

We thank Christian Mossin for graciously supplying us with the source code of his programs and sample inputs. Thanks are also due to the PEPM '95 referees who provided detailed and constructive comments. Frank Knoll was a patient beta tester during the development of his C front end. Finally, thanks are due to the anonymous reviewers who provided helpful comments. One suggested that partial evaluation might transform the direct-style parser into the continuation-based formulation; we are especially grateful for his insight.

REFERENCES

Bondorf, A. 1991. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming 17*, 3–34.

Bondorf, A. 1993. *Similix 5.0 Manual.* DIKU, University of Copenhagen.

Chapman, N. P. 1987. *LR parsing: theory and practice.* Cambridge University Press, Cambridge.

Chirica, L. and Martin, D. 1979. An order algebraic definition of Knuthian semantics. *Theor. Comp. Sci. 13*, 1–27.

Consel, C. and Danvy, O. 1991. For a better support of static data flow. See Hughes [1991], pp. 496–519.

Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, pp. 493–501. ACM Press.

Danvy, O. and Filinski, A. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science 2*, 361–391.

Danvy, O., Malmkjær, K., and Palsberg, J. 1996. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems 18,* 6 (Nov.), 730–751.

Dencker, P., Dürre, K., and Heuft, J. 1984. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems 6,* 4 (Oct.), 546–572.

DeRemer, F. and Pennello, T. 1982. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems 4,* 4 (Oct.), 615–649.

DeRemer, F. L. 1969. Practical translators for LR(k) parsers. Ph.D. thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma.

DeRemer, F. L. 1971. Simple LR(k) grammars. *Communications of the ACM 14,* 7, 453–460.

Donnelly, C. and Stallman, R. 1995. *Bison—The YACC-compatible Parser Generator.* Boston, MA: Free Software Foundation. Part of the Bison distribution.

Dybkjær, H. 1985. Parsers and partial evaluation: An experiment. Student Report 85-7-15 (July), DIKU, University of Copenhagen, Denmark.

Ershov, A. P. and Ostrovsky, B. N. 1987. Controlled mixed computation and its application to systematic development of language-oriented parsers. In L. G. L. T. Meertens (Ed.), *Program Specification and Transformation, Proc. IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, pp. 31–48. North Holland.

Fischer, M. J. 1993. Lambda-calculus schemata. *Lisp and Symbolic Computation 6,* 3/4, 259–288.

Futamura, Y. 1971. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls 2,* 5, 45–50.

Gomard, C. K. and Jones, N. D. 1991. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming 1,* 1 (Jan.), 21–70.

Grosch, J. 1990. Lalr—a generator for efficient parsers. *Software—Practice & Experience 20,* 11 (Nov.), 1115–1135.

Holst, C. K. and Gomard, C. K. 1991. Partial evaluation is fuller laziness. In P. Hudak and N. D. Jones (Eds.), *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, New Haven, CT, pp. 223–233. ACM. SIGPLAN Notices 26(9).

Hughes, J. (Ed.) 1991. *Functional Programming Languages and Computer Architecture*, Volume 523 of *Lecture Notes in Computer Science*, Cambridge, MA. Springer-Verlag.

Hutton, G. 1990. Parsing using combinators. In K. Davis and J. Hughes (Eds.), *Functional Programming, Glasgow 1989*, pp. 353–370. Springer-Verlag.

IEEE 1991. Standard for the Scheme programming language. Tech. Rep. 1178-1990, Institute of Electrical and Electronic Engineers, Inc., New York.

Ives, F. 1986. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices 21,* 7 (July), 131–135. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

Ives, F. 1987a. An LALR(1) lookahead set algorithm. Unpublished manuscript.

Ives, F. 1987b. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices 22,* 8 (August), 99–104.

Johnson, S. C. 1975. Yacc—yet another compiler compiler. Tech. Rep. 32, AT&T Bell Laboratories, Murray Hill, NJ.

Johnsson, T. 1985. Lambda lifting: Transforming programs to recursive equations. In *Proc. Functional Programming Languages and Computer Architecture 1985*, Volume 201 of *Lecture Notes in Computer Science.* Springer-Verlag.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall.

JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. 1985. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. JOUANNAUD (Ed.), *Rewriting Techniques and Applications*, Dijon, France, pp. 124–140. Springer-Verlag. LNCS 202.

KELSEY, R., CLINGER, W., AND REES, J. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11,* 1, 7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available electronically as `http://www.neci.nj.nec.com/homepages/kelsey/r5rs.ps.gz`.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language.* Prentice-Hall. 2nd edition.

KNUTH, D. E. 1965. On the translation of languages from left to right. *Information and Control 8*, 607–639.

KNUTH, D. E. 1968. Semantics of context-free languages. *Mathematical Systems Theory 2*, 127–145.

KNUTH, D. E. 1971. Semantics of context-free languages. *Mathematical Systems Theory 5*, 95–96. Correction to [Knuth 1968].

LAUNCHBURY, J. AND HOLST, C. K. 1991. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, Skye, Scotland, pp. 210–218. Glasgow University.

LAWALL, J. L. AND DANVY, O. 1994. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, USA, pp. 227–238. ACM Press.

LEERMAKERS, R. 1993. *The Functional Treatment of Parsing.* Kluwer Academic Publishers, Boston.

MOSSIN, C. 1993. Partial evaluation of general parsers. In D. SCHMIDT (Ed.), *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, Copenhagen, Denmark, pp. 13–21. ACM Press.

PAGAN, F. L. 1991. *Partial Computation and the Construction of Language Processors.* Prentice-Hall.

PARK, J. C. AND CHOE, K.-M. 1987. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices 22,* 4 (April), 30–32.

PARK, J. C. H., CHOE, K. M., AND CHANG, C. H. 1985. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems 7,* 1 (Jan.), 159–175.

PENNELLO, T. J. 1986. Very fast LR parsing. *SIGPLAN Notices 21,* 7, 145–151.

PLOTKIN, G. 1975. Call-by-name, call-by-value and the λ-calculus. *Theor. Comp. Sci. 1*, 125–159.

REYNOLDS, J. C. 1972. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pp. 717–740.

RÖHRICH, J. 1980. Methods for the automatic construction of error correcting parsers. *Acta Inf. 13*, 115–139.

ROBERTS, G. H. 1988. Recursive ascent: An LR analog to recursive descent. *SIGPLAN Notices 23,* 8, 23–29.

SIPPU, S. AND SOISALON-SOININEN, E. 1990. *Parsing Theory*, Volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin.

SPERBER, M. 1994. Attribute-directed functional LR parsing. Unpublished manuscript.

SPERBER, M. AND THIEMANN, P. 1999. *Essence—User Manual.* Freiburg, Germany: Universität Freiburg. Available from `ftp://ftp.informatik.uni-freiburg.de/iif/thiemann/essence/`.

SWIERSTRA, S. D. AND DUPONCHEEL, L. 1996. Deterministic, error-correcting combinator parsers. In J. LAUNCHBURY, E. MEIJER, AND T. SHEARD (Eds.), *Advanced Functional Programming*, Volume 1129 of *Lecture Notes in Computer Science*, pp. 184–207. Springer-Verlag.

TARJAN, R. E. AND YAO, A. 1979. Storing a sparse table. *Communications of the ACM 22,* 11, 606–611.

THIEMANN, P. 1996a. Cogen in six lines. In R. K. DYBVIG (Ed.), *Proc. International Conference on Functional Programming 1996*, Philadelphia, PA, pp. 180–189. ACM Press, New York.

Thiemann, P. 1996b. Towards partial evaluation of full Scheme. In G. Kiczales (Ed.), *Reflection'96*, San Francisco, CA, USA, pp. 95–106.

Thiemann, P. 1999. *The PGG System—User Manual.* Freiburg, Germany: Universität Freiburg. Available from `ftp://ftp.informatik.uni-freiburg.de/iif/thiemann/pgg/`.

Turchin, V. F. 1979. A supercompiler system based on the language Refal. *SIGPLAN Notices 14,* 2 (Feb.), 46–54.

Wilhelm, R. and Maurer, D. 1995. *Compiler Design.* Addison-Wesley.

Received August 1998; revised March 1999; accepted June 1999