A Formulation of Deterministic Bottom-Up Parsing and Parser Generation In Logic Programming

Chuck Liang

Department of Computer Science
Trinity College
300 Summit Street
Hartford, CT 06106-3100, USA
chuck.liang@mail.trincoll.edu

Abstract. This paper addresses efficient parsing in the context of logical inference for the purpose of using logic programming languages in practical compiler writing. A bottom-up, deterministic parsing mechanism is formulated for "bounded right context" grammars, a subclass of LR(k) grammars with characteristics amenable to declarative parser specification. A working parser generator for a logic programming language is described.

1 Introduction

The general aim of the work presented here is to use logic programming as a practical instrument for compiler writing and other activities concerning programming languages and systems. It is natural to expect that logic programming, with its algebraic syntax and unification algorithms, is suitable for formulating parsers for translating concrete syntax to abstract syntax. Parsing in Prolog has traditionally relied on definite clause grammars (DCG) [12]. In an environment that implements depth-first proof search, DCG's represent a top-down, recursive-descent parsing mechanism that suffers from inefficiencies caused by non-determinism and even non-termination. Some of these problems are alleviated by alternative implementations and optimizations of logic programming. They alone however, can not replace all efficient parsing strategies, such as the use of lookahead symbols and precedence functions. Although DCG's have been used in compiler writing, extensive grammar specialization and other techniques are generally required (e.g. to deal with left-recursion and associativity) before syntax can be parsed deterministically. Logic programming has scarcely taken advantage of the powerful bottom-up parsing technologies, particularly LR grammars and "yacc" style tools commonly used in compiler writing. One obvious remedy for this situation would be to implement the well-known LR parsing algorithms in Prolog using brute-force methods. Such an approach, however, would derive few advantages from the use of logic programming, and the resulting parser will little resemble its declarative grammar. Declarative programming views computation as logical deduction, and would benefit from studies of parsing in this context.

In this paper we resurrect a class of context free grammars that are closely related to LR grammars and are likewise capable of describing all deterministic context-free languages. Characteristics of these grammars are exploited in the formulation of a bottom-up parsing strategy in the framework of logical inference. A set of *parsing inference rules* are derived from a grammar that can be directly implemented as Horn clauses. These rules will be *deterministic* in the sense that at most one rule is applicable at any time, and *linear* in the sense that every rule has at most one recursive premise. They are also terminating. Efficient shift-reduce parsing is thus manifested as logic programming.

The general concept of bottom-up parsing as logical deduction presented here has some precedence in the work found in [13, 15] and especially [3]. We shall discuss the relationship of these works to ours in greater detail in Section 7.

2 Bounded Right Context Grammars

The following presentation of the theoretical foundations of this paper assumes a basic familiarity with bottom-up parsing concepts such as provided in compiler texts (e.g. [1, 2]). A more detailed introduction to the theory of deterministic languages and parsing can be found in [8].

We briefly state some basic definitions. A context-free grammar is a tuple (V, Σ, P, S), where V represents a finite set of grammar symbols and $\Sigma \subset V$ a set of "terminal" symbols. P is a finite set of "productions" of the form $A \to \gamma$ where $A \in N = V - \Sigma$ (the non-terminals) and $\gamma \in V^*$. $S \in N$ is the designated "start symbol." A *rightmost derivation* step is of the form $\alpha A w \Longrightarrow_r \alpha\beta w$ where $\alpha \in V^*$, $w \in \Sigma^*$ and $A \to \beta \ \in \ P$. Derivation in arbitrary numbers of steps are represented by $\stackrel{*}{\Longrightarrow}_r$ and $\stackrel{+}{\Longrightarrow}_r$. If $\gamma \stackrel{*}{\Longrightarrow}_r \epsilon$, the empty sequence, then γ is said to be *nullable*. We also confine our discussion to *reduced* context-free grammars where for all $A \in V$, $S \stackrel{*}{\Longrightarrow}_r \alpha A \beta \stackrel{*}{\Longrightarrow}_r w$ such that $w \in \Sigma^*$. In other words, all grammar symbols are reachable from the start symbol and every symbol derives a sequence of terminal symbols. We also exclude grammars where $A \stackrel{+}{\Longrightarrow}_r A$ is possible for any non-terminal A. Reduced grammars allowing such derivations are necessarily ambiguous.

If $S \stackrel{*}{\Longrightarrow}_r \gamma$ then γ is a *right sentential form* of the grammar. If $S \stackrel{*}{\Longrightarrow}_r \alpha A w$ and $\alpha A w \Longrightarrow_r \alpha\beta w$ then any prefix of $\alpha\beta$ is a *viable prefix,* and $A \to \beta$ is called the *handle* of $\alpha\beta w$ *at position* $\alpha\beta$*.* The handle identifies which production to apply in reverse (and where) in a bottom-up, rightmost derivation.

The type of grammar we consider suitable for formulating bottom-up parsing as deterministic deduction are known as *bounded right context* (BRC) grammars, introduced by Floyd [4]. The principle characteristic of a BRC grammar is that the unique handle of a bottom-up, right-most derivation step can be determined by looking *ahead* some k symbols to the right (the remaining input), and looking *back* some l symbols to the left (the stack). The better known $LR(k)$ grammars also require lookaheads of k symbols to the right, but looks back at the *entire* viable prefix to recognize handles. Implementations of LR parsers rely on deterministic finite state machines to keep track of the contents of the viable prefix[1].

Let $last_l(\gamma)$ be the length-l suffix of of γ (or γ if γ contains less than l symbols), and let $first_k(\gamma)$ be defined in the usual way: the set of length-k prefixes of terminal sequences derivable from γ (or length $k' < k$ sequences if γ derives a string of length k'). Formally, a context free grammar (V, Σ, P, S) is of type $BRC(l, k)$ if the following condition holds (from [7]):

1. $S \overset{*}{\Longrightarrow}_r \alpha A w \Longrightarrow_r \alpha\beta w$,
2. $S \overset{*}{\Longrightarrow}_r \alpha_2 A_2 w_2 \Longrightarrow_r \alpha_2\beta_2 w_2 = \sigma\beta u$ such that $\sigma\beta$ is a prefix of $\alpha_2\beta_2$, and
3. $first_k(w) = first_k(u)$ and $last_l(\alpha) = last_l(\sigma)$

implies

$$A \rightarrow \beta \;=\; A_2 \rightarrow \beta_2 \quad \text{and} \quad \alpha_2\beta_2 = \sigma\beta.$$

If we restrict the preconditions of the definition so that $\alpha = \sigma$ (which would also make redundant the "lookback" condition $last_l(\alpha) = last_l(\sigma)$), then this definition would be equivalent to the one for $LR(k)$ grammars ([9]). Thus it is easy to see why every $BRC(l, k)$ grammar is immediately an $LR(k)$ grammar. Furthermore, it has been proved ([9, 7]) that BRC grammars are capable of describing the same set of languages as LR grammars (namely all deterministic context-free languages). In particular, every $LR(k)$ grammar has an equivalent $BRC(1, k)$ grammar. In fact, we find that many examples of LR grammars are already also of type BRC[2]. Development of BRC strategies for parsing programming language syntax was apparently halted due to the advent of efficient LR parsing. Most implementations of LR parsing are for the SLR and LALR subclasses of $LR(k)$ grammars. The BRC subclass is of an entirely different nature, and it is not our intention to compete with other parsing algorithms. Instead, we show that BRC grammars allow efficient and useful parsers to be specified as logical inference rules. Such a specification would be far more costly for general LR grammars due to the need to compute and represent state information. That is, the simplification afforded by BRC grammars can be exploited to formulate a *proof-theoretic* parsing mechanism that can take advantage of the declarative syntax and unification capabilities of logic programming.

The basic logical framework of our parsing mechanism is similar to the "parsing as deduction" principle given in [13, 15] (also used in [3]). We introduce a "parsing judgement" of the form $\alpha \triangleleft w$, where α is a sequence of grammar symbols and w a sequence of terminal grammar symbols. The intended meaning of the judgement is that α is a viable prefix of right sentential form αw of the

[1] In other words, the lookback, unlike the lookahead, can not just be a finite sequence of grammar symbols, but must be a regular expression if we are to algebraically specify the $LR(k)$ handles.

[2] All grammars used to illustrate LR grammars in [1] and [2] are in fact $BRC(1, 1)$ or simpler grammars.

grammar under consideration. Intuitively, α represents the parse stack and w the remaining input.

Bottom-up right-most derivation is represented by the following types of inference rules (which are meant to be read "bottom-up"):

$$\frac{\alpha\beta_l A \triangleleft a_k w}{\alpha\beta_l \gamma \triangleleft a_k w}\ \mathit{Reduce} \qquad \text{and} \qquad \frac{\alpha\beta_l\gamma a \triangleleft w}{\alpha\beta_l\gamma \triangleleft aw}\ \mathit{Shift}.$$

The sequence β_l represents a "lookback" of l symbols, and a_k represents a "lookahead" of k terminal symbols. Both l and k can be 0. In the reduce rule, $A \to \gamma$ is a production of the grammar, while in the shift rule γ is a non-ϵ proper prefix of some right hand side $\gamma B \gamma'$ of a production such that $B \in V$ is non-nullable and $a \in \mathit{first}(B)$. The first set (or relation) can be pre-computed so this is not necessarily a costly condition to verify (especial since in actual implementation all shift-rules can be collapsed into one default rule). These rules can be directly encoded as Horn clauses with the bottom judgement at the head of each clause.

Non-determinism among these rules may appear between two reduce rules, or between a reduce and a shift rule whose bottom sides share a common instance. For BRC grammars, sufficient lookbacks and lookaheads can be computed to eliminate all non-determinism from their inference rules. Our formulation here is generalized to $BRC(l,k)$ grammars. In practice, however only one lookback and one lookahead symbol are usually needed. It follows directly from the results established in [9, 7] that every deterministic language has a $BRC(1,1)$ grammar.

3 Valid Handles and Inference Rules

The symmetrical nature of BRC grammars suggests that we augment each grammar with an implicit production $S' \to \$_0 S \$_1$, where $\$_0$ and $\$_1$ are unique symbols representing "begin of file" and "end of file" respectively. The $\$_0$ symbol ensures that lookback sequences are never empty.

We define a *contexted handle* of a grammar to be a triple $(\beta_l, A \to \gamma, a_k)$ such that $A \to \gamma$ is a production, β_l are either l grammar symbols or $l' < l$ symbols beginning with $\$_0$, and a_k are either k terminal symbols or $k' < k$ terminal symbols ending in $\$_1$. Contexted handles (with fixed l and k) are "valid" if they satisfy the following:

Definition 1. (*valid* (l,k) *handles*)
Given a grammar (V, Σ, P, S) augmented with $S' \to \$_0 S \$_1$,

1. $(\$_0, S \to \gamma, \$_1)$ is a valid handle for each production $S \to \gamma$.
2. if $(\beta_l, A \to \sigma B \gamma, a_k)$ is a valid handle, $\beta_l' = \mathit{last}_l(\beta_l \sigma)$, and $a_k' \in \mathit{first}_k(\gamma a_k)$, then $(\beta_l', B \to \rho, a_k')$ is a valid handle for each production $B \to \rho$.

We emphasize that unlike first_k, last_l need not be a set since the lookback may contain non-terminal as well as terminal symbols.

Valid handles characterize the right sentential forms of a grammar up to a "bounded context," as formalized by the following lemma (it's implied here that $\alpha = \epsilon$ if β_l begins with $\$_0$, and analogously for w):

Lemma 2. *$(\beta_l, A \to \gamma, a_k)$ is a valid (l,k) handle of a grammar if and only if $S' \stackrel{*}{\Longrightarrow}_r \alpha\beta_l A a_k w$ for some $\alpha \in V^*$ and $w \in \Sigma^*$.*

Proof: by induction on the length of rightmost derivations and the definition of valid handles. □

The computation of valid handles is similar to the computation of the "follow" set described in compiler texts, except that the left and right contexts of non-terminal symbols are kept tract of simultaneously. That is, we can compute the handles by starting with $(\$_0, S \to \gamma, \$_1)$ and follow the above inductive definition until a closure is formed (the computation is also bounded by l, k and the number symbols in the grammar). We shall describe the formation of valid handles further in Section 5.

A grammar with a set of valid handles gives rise to a set of *canonical* inference rules as defined below.

Definition 3. (*Canonical Inference Rules*)
Given the valid (l,k) handles of a grammar:

- The implicit production $S' \to \$_0 S \$_1$ yields two special rules:

$$\frac{\$_0 a \lhd a'_{k-1} w}{\$_0 \lhd a a'_{k-1} w}\ \mathit{Shift}_{,\, aa'_{k-1} \in first_k(S)}, \quad \text{and} \quad \frac{}{\$_0 S \lhd \$_1}\ \mathit{Accept}$$

- For each valid handle $(\beta_l, A \to \gamma, a_k)$ there exists a rule

$$\frac{\alpha\beta_l A \lhd a_k w}{\alpha\beta_l \gamma \lhd a_k w}\ \mathit{Reduce}$$

  Here, α and w are schematic variables ranging over arbitrary sequences of grammar symbols and terminal grammar symbols respectively.
- For each valid handle $(\beta_l, A \to \gamma, b_k)$ where $\gamma = G_1 \ldots G_n$ for $G_1, \ldots, G_n \in V$ and $n > 1$, and for each i such that $1 \leq i < n$ where G_{i+1} is not nullable, there exists a rule[3]

$$\frac{\alpha\beta_l G_1 \ldots G_i a \lhd a'_{k-1} w}{\alpha\beta_l G_1 \ldots G_i \lhd a a'_{k-1} w}\ \mathit{Shift}_{,\, aa'_{k-1} \in first_k(G_{i+1} \ldots G_n b_k)}$$

  Here, if $k = 0$ (no lookahead) then the $first_k$ side condition is omitted as long as $a \neq \$_1$. If $k = 1$ (one lookahead) then, since G_{i+1} is non-nullable, the side condition simplies to $a \in first(G_{i+1})$.

[3] It can be shown that G_{i+1} is nullable only if $G_{i+1} \stackrel{*}{\Longrightarrow}_r B\sigma$ such that $B \to \epsilon$ is a production and σ is nullable. But then the valid handles for $B \to \epsilon$ will inherit the same lookback and lookahead symbols from G_{i+1}. Adding a shift rule at this point will thus result in a shift-reduce conflict with $B \to \epsilon$.

When $k > 1$, it is necessary that shift rules, while shifting only one symbol at time, must still check that the next k symbols on the right of $\triangleleft$ are derivable from the remainder of the production plus the lookahead of the valid handle. Otherwise the absence of "shift-reduce" conflicts would be meaningless.

Formally, A *reduce-reduce* conflict between two distinct canonical inference rules exists if they are of the forms

$$\frac{\alpha\beta_l A \triangleleft a_k w}{\alpha\beta_l\gamma \triangleleft a_k w}\ \mathit{Reduce} \qquad \text{and} \qquad \frac{\alpha'\beta_l' A' \triangleleft a_k w'}{\alpha'\beta_l'\gamma' \triangleleft a_k w'}\ \mathit{Reduce}$$

such that $\alpha\beta_l\gamma = \alpha'\beta_l'\gamma'$ for some α and α'. This condition is easily checked by unification where α and α' are free variables. For example, a reduce-reduce conflict exists between one rule having bottom side αaaA and another one have $\alpha' aA$, since $\alpha' = \alpha a$ is possible.

Similarly, a *shift-reduce* conflict exists between any two rules of the forms

$$\frac{\alpha\beta_l A \triangleleft aa'_{k-1} w}{\alpha\beta_l\gamma \triangleleft aa'_{k-1} w}\ \mathit{Reduce} \qquad \text{and} \qquad \frac{\alpha'\beta_l'\gamma' a \triangleleft a'_{k-1} w'}{\alpha'\beta_l'\gamma' \triangleleft aa'_{k-1} w'}\ \mathit{Shift}$$

if $\alpha\beta_l\gamma = \alpha'\beta_l'\gamma'$ for some α and α'.

We say that a set of canonical inference rules are *deterministic* if there exists no shift-reduce or reduce-reduce conflicts.

3.1 Examples

The correctness of this parsing scheme is addressed in section 4, but first we give some examples of grammars and their encoding as bottom-up inference rules.

The following simple grammar requires a single lookback symbol for determinism:

$S' \rightarrow \$_0 A \$_1$
$A \rightarrow Aa \mid a$

This $BRC(1,0)$ (and $LR(0)$) grammar has the valid handles (the lookahead components are omitted):

- $(\$_0, A \rightarrow Aa)$
- $(\$_0, A \rightarrow a)$

These in turn give rise to the following deterministic inference rules

$$\frac{\$_0 A \triangleleft w}{\$_0 Aa \triangleleft w}\ R \quad \frac{\$_0 A \triangleleft w}{\$_0 a \triangleleft w}\ R \quad \frac{\$_0 At \triangleleft w}{\$_0 A \triangleleft tw}\ S \quad \frac{\$_0 t \triangleleft w}{\$_0 \triangleleft tw}\ S \quad \frac{}{\$_0 A \triangleleft \$_1}\ A$$

Without the lookback symbol $\$_0$, there would exist a reduce-reduce conflict between the productions $A \rightarrow Aa$ and $A \rightarrow a$. That is, if Aa existed on top of the stack it could be reduced to either A or AA, the latter of which is invalid. Note also that, since no lookaheads are needed, the shift rules need not consider which symbol is being shifted (as long as it's not $\$_1$).

The following similar grammar is of type $BRC(0,1)$ (and thus LR(1)):

$S' \to \$_0 A \$_1$
$A \to aA \ | \ a$

The inference rules from the analogous valid handles $(A \to aA, \$_1)$ and $(A \to a, \$_1)$ are

$$\frac{\alpha A \triangleleft \$_1}{\alpha a A \triangleleft \$_1}\ R \quad \frac{\alpha A \triangleleft \$_1}{\alpha a \triangleleft \$_1}\ R \quad \frac{\alpha a a \triangleleft w}{\alpha a \triangleleft a w}\ S \quad \frac{\$_0 a \triangleleft w}{\$_0 \triangleleft a w}\ S \quad \frac{}{\$_0 A \triangleleft \$_1}\ A$$

In this case the *lookahead* symbol $\$_1$ prevented a shift-reduce conflict between

$$\frac{\alpha A \triangleleft \$_1}{\alpha a \triangleleft \$_1}\ R \quad \text{and} \quad \frac{\alpha a a \triangleleft w}{\alpha a \triangleleft a w}\ S,\ a \in first(A).$$

Since lookaheads are used the shift rule must ensure that the symbol being shifted belonged to $first(A)$ (which in this case is just a).

The following grammar (with implicit production for S' omitted) requires both a lookback and a lookahead and is of type $BRC(1,1)$. The lack of either would lead to reduce-reduce conflicts between $A \to c$ and $B \to c$ and a non-deterministic parser:

$S \ \to \ aAd \ | \ aBe \ | \ bAe \ | \ bBd$
$A \ \to \ c$
$B \ \to \ c$

With both lookback and lookahead, non-conflicting reduce rules for the A and B productions are derived:

$$\frac{\alpha a A \triangleleft dw}{\alpha a c \triangleleft dw}\ R, \quad \frac{\alpha a B \triangleleft ew}{\alpha a c \triangleleft ew}\ R, \quad \frac{\alpha b A \triangleleft ew}{\alpha b c \triangleleft ew}\ R, \quad \frac{\alpha b B \triangleleft dw}{\alpha b c \triangleleft dw}\ R$$

The next example shows that a grammar can be $LR(k)$ but not BRC:

$S \to aA \ | \ bB$
$A \to (A) \ | \ c$
$B \to (B) \ | \ c$

No lookback of any fixed length can resolve all conflicts between $A \to c$ and $B \to c$. However, this $LR(0)$ grammar is equivalent to a $BRC(0,0)$ grammar by eliminating the B productions:

$$S \to aA \ | \ bA, \qquad A \to (A) \ | \ id$$

Alternatively, one can rewrite the grammar so that the *state* information as to whether a or b was seen first is carried by special non-terminal symbols[4]:

[4] A general procedure for transforming $LR(k)$ grammars into $BRC(1,k)$ grammars exists (see [6]). However, this procedure assumes that the state machine of a LR parser is already available. In most cases, however, the necessary modifications are of the kind shown in this example.

$S \to aA \mid bB$
$A \to PA) \mid id$
$B \to QB) \mid id$
$P \to ($
$Q \to ($

This example also demonstrates how lookbacks are more powerful than lookaheads, which can consist of only terminal symbols. The non-terminals P and Q distinguish "(" symbols associated with A productions from those associated with B productions (as well as the P and Q productions themselves). The valid handles handles (with one lookback) are:

- $(a, A \to PA))$ $(a, A \to id)$
- $(P, A \to PA))$ $(P, A \to id)$
- $(b, B \to QB))$ $(b, B \to id)$
- $(Q, B \to QB))$ $(Q, B \to id)$
- $(a, P \to ()$ $(P, P \to ()$
- $(b, Q \to ()$ $(Q, Q \to ()$

This is a $BRC(1, 0)$ grammar. It is also a "simple BRC grammar," which we define in Section 5.1.

4 Correctness

The formal correctness of our deductive parsing mechanism consists of lemma 2 plus the following results. We use "deduction" to represent the application of inference rules and "derivation" to mean rightmost derivations of grammar symbols.

Lemma 4. *If there is a deduction of $\alpha \triangleleft w$ from $\$_0 S \triangleleft \$_1$ then α is a viable prefix and αw is a right sentential form of the grammar.*

Proof: by induction on the height of deductions, appealing to lemma 2. □

Lemma 5. *If $S' \overset{*}{\Longrightarrow}_r \alpha A w$, then there is a deduction from $\$_0 S \triangleleft \$_1$ of $\alpha A \triangleleft w$*

Proof: by induction on the length of rightmost derivations, appealing to lemma 2. □

These "soundness and completeness" lemmas are better understood *top-down,* although they are meant to establish the correctness of a bottom-up parsing strategy, formalized by the following theorem:

Theorem 6. *A grammar is $BRC(l, k)$ if and only if its canonical inference rules are deterministic.*

Proof: The forward direction is proved by contradiction. Using the previous lemmas, it is seen that a reduce-reduce conflict entails the existence of two right sentential forms that satisfy the preconditions of the definition of BRC grammars but contradict the requirement that $A \rightarrow \beta = A_2 \rightarrow \beta_2$. Similarly, a shift-reduce conflict contradicts the requirement that $\alpha_2\beta_2 = \sigma\beta$: $\sigma\beta$ will remain a *proper* prefix of $\alpha_2\beta_2$. For the reverse direction, it can be shown that if $\sigma\beta$ is a proper prefix of $\alpha_2\beta_2$ then at some earlier point in the bottom-up inference of a terminal sequence the same reduce rule was applicable, and by determinism will entail a different sentential form from $\alpha_2\beta_2 w_2$[5]. Once we know that $\alpha_2\beta_2 = \sigma\beta$, $A \rightarrow \beta = A_2 \rightarrow \beta_2$ follows directly from the absence of reduce-reduce conflicts. □

Termination of bottom-up deductions is complicated by the presence of ϵ-productions (since they can expand the size of the stack). It can be shown that termination for grammars without ϵ-productions does not require determinism (because we assume that $A \stackrel{+}{\Longrightarrow}_r A$ is not possible). However, it's easily seen how, without any lookbacks or lookaheads, any grammar with an ϵ-production has an infinite sequence of deductions in reverse. With ϵ-productions, determinism is a necessary condition for termination:

Theorem 7. *There are no infinite sequences of deductions in reverse for the canonical inference rules of a $BRC(l,k)$ grammar.*

Proof: It suffices to show that there can not be an infinite sequence of reduce steps between shifts. Since reduce rules derive from valid handles, if there is such an infinite sequence then, using lemma 2, it can be shown that there is also an infinite sequence of reductions in reverse for some right sentential form. This contradicts determinism and lemma 5. □

5 Handle Merging

Two valid handles of the form $(b, A \rightarrow \gamma, a)$ and $(b, A \rightarrow \gamma, a')$ can be merged to form one set of inference rules without introducing new conflicts, since either lookahead will cause the same action. The merged handle would be

$$(b, A \rightarrow \gamma, \{a, a'\}).$$

Handles of the form $(b', A \rightarrow \gamma, a)$ and $(b, A \rightarrow \gamma, a)$ can similarly be merged safely. Handles of the form $(b, A \rightarrow \gamma, a)$ and $(b', A \rightarrow \gamma, a')$, however, can not in general be merged without causing new conflicts (consider the properly $BRC(1,1)$ grammar of section 3.1, for example). In the representation of valid handles we can therefore use sets for both the lookbacks and lookaheads. Valid handles of a production with identical lookbacks can first be merged, then valid handles with the same lookahead sets can be merged as well. Handle merging

[5] This argument uses the fact that in reduced grammars all sentential forms derive terminal sequences, and that grammars where $A \stackrel{+}{\Longrightarrow}_r A$ is possible are excluded.

is an important implementation-wise feature because it significantly reduces the number of inference rules needed.

For example, the following $BRC(1,1)$ grammar describes "assignment statements" in C:

$$
\begin{aligned}
&S' \to \$_0 S \$_1 \\
&S \to L = R \ | \ R \\
&L \to *R \ | \ id \\
&R \to L
\end{aligned}
$$

There are sixteen valid handles of this grammar, but they can be safely merged into the following eight handles:

- $(\{\$_0\}, S \to L = R, \{\$_1\})$, $(\{\$_0\}, S \to R, \{\$_1\})$
- $(\{*, \$_0\}, L \to *R, \{=, \$_1\})$, $(\{*, \$_0\}, L \to id, \{=, \$_1\})$
- $(\{=\}, L \to *R, \{\$_1\})$, $(\{=\}, L \to id, \{\$_1\})$
- $(\{*\}, R \to L, \{\$_1, =\})$, $(\{\$_0, =\}, R \to L, \{\$_1\})$

These merged handles give rise to inference rules with extra side conditions, such as

$$\frac{\alpha b L \lhd aw}{\alpha b * R \lhd aw} \ R, b \in \{*, \$_0\}, a \in \{=, \$_1\})$$

Note that the *union* of all lookaheads of the valid handles of $L \to id$ and $L \to *R$ is exactly the "follow" set of L, and similarly for S and R. We exploit this characteristic in the next section.

5.1 Simple BRC

For $BRC(1,1)$ grammars, the number of inference rules resulting from the (merged) valid handles is comparable to the number of states in the viable-prefix finite automaton of a $LR(1)$ parser[6]. The previous section suggests a technique that is analogous to "SLR" parsing in relation to $LR(1)$ parsing. We allow valid handles of the form $(b, A \to \gamma, a)$ and $(b', A \to \gamma, a')$ to be merged into $(\{b, b'\}, A \to \gamma, \{a, a'\})$. This will ensure that the number of merged handles is always the same as the number of productions of the grammar (minus the initial production for S'). Conflicts introduced by this type of merging can still be detected and resolved by other means (e.g., the user can be prompted to choose which action should be given preference). We say that a grammar is a *Simple BRC* (SBRC) grammar if the resulting set of inference rules derived from the "simply-merged" handles remain deterministic. The following grammar is often used as a standard example of bottom-up parsing:

$$
\begin{aligned}
&S' \to \$_0 E \$_1 \\
&E \to E + T \ | \ T \\
&T \to T * F \ | \ F \\
&F \to (E) \ | \ id
\end{aligned}
$$

[6] Each rule roughly corresponds to an occurrence of a "kernel item" in a DFA state.

The fully merged valid handles of this grammar are:

- $(\{\$_0, (\}, E \rightarrow E+T, \{\$_1, +,)\})$, $(\{\$_0, (\}, E \rightarrow T, \{\$_1, +,)\})$
- $(\{\$_0, (, +\}, T \rightarrow T*F, \{\$_1, +, *,)\})$, $(\{\$_0, (, +\}, T \rightarrow F, \{\$_1, +, *,)\})$
- $(\{\$_0, (, +, *\}, F \rightarrow (E), \{\$_1, +, *,)\})$, $(\{\$_0, (, +, *\}, F \rightarrow id, \{\$_1, +, *,)\})$

The inference rules derived from these handles remain deterministic. Note in particular the absence of $+$ among the lookbacks of the handles for $E \rightarrow T$, and the absence of $*$ from the lookbacks for $T \rightarrow F$. This ensures that $E+T$ (or $T*F$) on top of the stack would not be reduced erroneously to $E+E$ (or $T*T$). The lookaheads of each handle is exactly the $follow$ set of the left-hand side non-terminal symbol of the associated production. The lookbacks of each handle would correspond to a left-side analogy of $follow$ (which we call the *before set*). Thus a SBRC parser generator need only compute the first, follow, and before relations in order to generate the merged handles from which inference rules can be derived. Experiments have shown that this technique, combined with operator precedence and associativity declarations and user-resolved shift-reduce conflicts, suffices to yield a useful parser generator for many purposes.

6 A Parser Generator

We now describe aspects of a working parser generator in a logic programming language[7]. Both the full $BRC(1,1)$ and the simple BRC methods have been implemented. The SBRC method has sufficed for most examples we've tried so far, and is currently being further developed. The programming language is λ*Prolog* (*Teyjus* implementation [10]). The choice of this language has to do with the semantic actions of a parser (they can generate "higher-order abstract syntax"), and not with any aspect of our parsing mechanism in particular. λProlog properly embeds Horn Clause Prolog. Notationally, application in λProlog is written in curried form: `(f a b)` instead of `f(a,b)`.

The input to the parser generator is itself a λProlog file (module) where a grammar is declared by a clause such as the following:

```
cfg   % the attributed context free grammar, with actions
[
  rule ((ae R) ==> [iconst R2]) (R is R2),
  rule ((ae R3) ==> [lparen,(ae R1),rparen]) (R3 is R1),
  rule ((ae R4) ==> [(ae A4),plust,(ae B4)]) (R4 is (A4 + B4)),
  rule ((ae R5) ==> [(ae A5),minust,(ae B5)]) (R5 is (A5 - B5)),
  rule ((ae R6) ==> [(ae A6),timest,(ae B6)]) (R6 is (A6 * B6)),
  rule ((ae R7) ==> [(ae A7),dividet,(ae B7)]) (R7 is (A7 div B7)),
  rule ((ae R8) ==> [(ae A8),expt,(ae B8)]) (power A8 A8 B8 R8)
].
```

[7] The full implementation, which also includes a semi-universal lexical analyzer, plus several larger examples and additional notes are available from the homepage at *http://www2.trincoll.edu/~cliang/parsergen*, or by contacting the author.

The implicit production for S' is internal. The symbols `ae, iconst, minust, plust, lparen`, etc.. are grammar symbols. A grammar symbol can be a function symbol with distinct variables, representing semantic attributes, as its arguments. Each production is represented by a `rule` term with the non-terminal on the left of the declared infix symbol `==>` and a list of symbols on the right. The last component of a `rule` term is a semantic action in the form of a λProlog goal, predicates for which (as in `power`) are defined in the same file (module). The format is therefore similar to yacc-style input files, except that there need not be a separate parser defined for the grammar specification. The current implementation uses the non-logical feature that when a logic variable is written to a file and reloaded, it is generalized into a universally quantified variable (similar to the behavior of `assert`). This feature simplifies the implementation, and as a consequence it requires that no variable in a list of grammar rules appears more than once, except in the semantic action goals. The argument to each functional grammar symbol in the grammar declaration also *must* be a variable. Fortunately most of these restrictions are necessary anyway, since otherwise, as with DCG's, the grammar may no longer be context *free*. This non-logical feature and its unwanted consequences can be eliminated by defining a separate parser and processor for grammar specifications, as in yacc.

Valid handles are generated by forming a closure of triples following the inductive definition as described in the foregoing. The generation of the $first$ relation needed for computation of lookaheads is stratified in stages, since no non-terminal symbol can produce a leading terminal by going through more than $n-1$ other productions where n is the total number of non-terminals. The detection of reduce-reduce and shift-reduce conflicts proceeds directly from the valid handles (except we consider each "`member`" of the lookbacks and lookaheads of a merged handle by virtue of Prolog's backtracking using negation-as-failure).

The generated representation of parsing judgements are four-place predicates of the form `parse Stack Input Result Rule_Type` where `Stack` and `Input` are lists, and `Rule_Type` is a string that's either `"shift", "reduce", "accept", "special"`, or `"error"`. `Result` is instantiated by the `"accept"` rule to the start symbol of the grammar along with its attribute (usually the abstract syntax tree). For example, a merged handle such as $(\{b\}, p \rightarrow qr, \{a1, a2\})$ can be represented by the clauses

```
parse [q,b|Stack] [r|Input] Result "shift" :-
  parse [r,q,b|Stack] Input Result Nextrule.
parse [r,q,b|Stack] [A|Input] Result "reduce" :-
  member A [a1,a2], (semantic_action),
  parse [p,b|Stack] [A|Input] Result Nextrule.
```

Semantic action goals are added to reduce rules in the obvious way. Our parser generator can easily be reconfigured to produce regular Prolog code as opposed to λProlog code. Output of parser code for other types of declarative languages may also be possible.

6.1 Operator Precedence and Associativity

The grammar shown in the `cfg` clause, used in an implementation of an "online calculator" however, is ambiguous as defined and the parser generator outputs the messages[8]:

```
Computing Valid Handles...
No reduce-reduce conflicts.
**Shift-Reduce conflict with  ae _63 ==> ae _85 :: plust :: ae _87
:: nil  exists
 when top of stack is of form  ae _683 :: plust :: ae _694 ::
bofs :: _68306
 and lookahead symbol is  plust
Do you want to (s)hift or (r)educe (default is shift):
```

As with yacc-style generators, shift-reduce conflicts caused by ambiguity concerning operator precedence and associativity can be resolved by supplementary declarations. This feature is easily incorporated into our parser generator by clauses illustrated by the following examples:

```
binaryop plust (ae X) (ae Y) "left" 3.
binaryop timest (ae X) (ae Y) "left" 2.
```

These clauses declare which grammar symbols are to be regarded as operators (on certain kinds of expressions) as well as their associativity and precedence level. Ambiguity caused by these operators are then resolved by `"special"` clauses at the beginning of the file. This feature requires the use of `!` in the parser (otherwise non-determinism due to backtracking will become possible). For example, Binary operator associativity is resolved by the following clause, which redirects a goal to a shift or reduce rule:

```
parse [Ea,OP,Eb|Alpha] [OP|Beta] Result "special" :-
  binaryop OP Ea Eb Assoc Prec, !,
  ( (Assoc = "left",
     parse [Ea,OP,Eb|Alpha] [OP|Beta] Result "reduce");
    (Assoc = "right",
     parse [Ea,OP,Eb|Alpha] [OP|Beta] Result "shift")).
```

Unary and implicit operators are handled similarly. A default `"error"` clause is also placed at the end of the file to report failure.

The parser displays messages for remaining conflicts as illustrated above. Reduce-reduce conflicts result in failure. Remaining shift-reduce conflicts are resolved by user input, producing `"special"` clauses that redirect goals matching the form of the conflicts to `"shift"` or `"reduce"` rules.

Another simplification (currently unimplemented) is to collapse all shift rules into a simple default shift rule, which is used only if no reduce rule is applicable.

[8] Symbols such as `_85` are internally generated logic variables. "`::`" is alternative notation for the list constructor `|`. `bofs` is $\$_0$.

This simplification however, means that a shift is always possible and errors will not be reported until the end of the file has been reached. This problem can be largely solved if the default shift rule is aware of the longest right-hand side of the grammar productions (which limits the number of symbols that can be shifted before a reduce rule must be applied).

We are also experimenting with a simple method that may allow our basic mechanism to parse a large number of properly LR grammars without undue overhead. It may be possible to automatically identify a class of terminal symbols which can resolve reduce-reduce conflicts if the parser is aware that they have been read.

7 Related Work

The concept of parsing as deduction was introduced by Warren and Pereira [13]. A recent work [15] extends their basic framework to various grammar formalisms. However, their formulations of parsing for context free grammars, including Earley's algorithm, remain essentially non-deterministic. Our parsing scheme can be seen as a specialization of their framework to parsing as *deterministic* deduction (and for deterministic languages).

A few other authors have addressed the use of bottom-up parsing in logic programming. SLR parsing is implemented directly in [11] and [14]. LR parsers have also been used (e.g. [5]) for natural language processing. A close precedent to the type of work presented here, however, is that of Cohen and Hickey [3], who described strategies for parsing in Prolog with similar goals. They showed how to compute grammar properties such as "first" and "follow" succinctly, and gave enough details for building LL(1) parsers in Prolog. Their framework for coding shift-reduce parsing is similar to ours. A formulation of deterministic bottom-up parsing was proposed based on "weak-precedence grammars." However, this formulation is incomplete: an extra condition that excludes productions with right-hand sides ending in the same symbol must be enforced, otherwise non-determinism of the reduce-reduce type may persist[9]. This extra condition however, further restricts the type of grammars that can be used with their technique. Nevertheless, the ultimate aims of this paper is very much in the same spirit as [3].

8 Conclusion

We have formulated a simple yet general technique for deterministic context free parsing in the context of logical inference. The Horn clause representation of parsers is adaptable to most logic programming settings.

Our parser generator has been used in the implementation of a Basic-like imperative language (with static scope and types). A parser for most of λProlog

[9] Consider the grammar $(S \rightarrow aB \mid cA, \quad A \rightarrow ab, \quad B \rightarrow b)$ in relation to the formulation of weak-precedence parsing in [3].

itself was also generated. An experimental interpreter for core SML in λProlog is currently being developed. Additionally, it has been used for purposes other than compiling as part of an interactive theorem prover for translating the syntax of various formal "object-logics" into λProlog host syntax. For future work we plan to investigate possible extensions to our technique. Implementation-wise we plan to improve the performance and usability of our parser generator, especially in the form of additional `"error"` clauses for error correction.

Acknowledgments

Support for this work was provided by Dale Miller at the Pennsylvania State University. The author also wishes to thank Dale Miller for helpful comments.

References

1. A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
3. J. Cohen and T. Hickey. Parsing and compiling using prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, 1987.
4. R. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.
5. S. Fong. *Computational Properties of Principle-Based Grammatical Theories.* PhD thesis, MIT, June 1991.
6. S. L. Graham. On bonded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254, 1974.
7. M. Harrison and I. Havel. On the parsing of deterministic languages. *Journal of the ACM*, 21(4):525–548, 1974.
8. M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.
9. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
10. G. Nadathur and D. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In *Automated Deduction–CADE-13*, Springer-Verlag LNAI no. 1632, pages 287–291, July 1999.
11. U. Nilsson. AID: an alternative implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.
12. F. Pereira and D. Warren. Definite clause grammars for language analysis. *Artificial Intelligence*, 13:231–278, 1980.
13. F. Pereira and D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.
14. D. Rosenblueth and J. Peralta. LR inference: Inference systems for fixed-mode logic programs, based on LR parsing. In *Proceedings of the 1994 International Symposium on Logic Programming*, pages 439–453, 1994.
15. Shieber S., Schabes Y., and Pereira F. Principles and implementation of deductive parsing. Technical Report TR-11-94, Center for Research in Computing Technology, Harvard University, December 1998.