A Deterministic Shift-Reduce Parser Generator for a Logic Programming Language

Chuck Liang

Department of Computer Science, Trinity College
300 Summit St., Hartford, CT 06106-3100, USA
email: 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 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 λProlog is described, although the basic parsing mechanism is applicable to logic programming in general.

1 Introduction

The overall aim of this paper is to use logic programming as a practical instrument for compiler writing and other activities concerning programming languages and systems. Many declarative methods in logic programming (e.g., higher order abstract syntax) have been developed for the representation and analysis of programming language constructs. However, logic programming is still not as widely used as conventional languages in compiler writing. One reason for this has been the lack of general and efficient parsing schemes. Parsing in Prolog has traditionally focused on a range of issues broader than programming languages (e.g., natural language processing). In such a context, generality often takes precedence over determinism. The well-known definite clause grammars (DCG, [13]) have a wide range of application. In a system using depth-first proof search however, DCGs represent a top-down, recursive-descent parsing mechanism that suffers from non-determinism and non-termination. Some of these problems are alleviated by alternative implementations and optimization techniques of logic programming. They alone, however, can not replace all efficient parsing strategies, such as the use of lookahead symbols and precedence functions. Although DCGs 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.

Bottom-up parsing in Prolog has also been studied. In fact, shift-reduce parsing of *any* context free grammar can be formulated by the following pair of rules (which are meant to be read bottom-up):

$$\frac{\alpha t \lhd w}{\alpha \lhd tw}\ Shift \quad \text{and} \quad \frac{\alpha A \lhd w}{\alpha\gamma \lhd w}\ Reduce,\ A \to \gamma \text{ is a production}$$

The $\triangleleft$ symbol separates the parse stack from the remaining input. α and w are schematic variables representing sequences of symbols[1]. These rules can be directly encoded as Horn Clauses with the bottom sides at the head of each clause. However, this "universal parser" will behave non-deterministically for all but the simplest grammars. Similar formulations of Earley's Algorithm also exist (see [14, 15]). Even LR-style parsing in Prolog has some precedents (e.g., [5, 12]). The aims of most such efforts, however, are again not specific to compiler construction (with the noted exception of [3] - see Section 7 for further discussion). As a consequence, these works are usually not concerned with the exact non-deterministic *choice points* of a grammar, namely reduce-reduce and shift-reduce conflicts. For example, in [12] non-determinism was *intentionally* preserved in order to process a larger class of logic programs. Unambiguous parser generation as required in compiler writing requires that non-deterministic choice points be identified and resolved.

One obvious solution to providing a deterministic parser for logic programming would be to directly implement the LR parsing algorithms described in compiler texts 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. Furthermore, this type of approach may not necessarily preserve all the advantages of LR parsing (e.g., "fast table lookup" may have no meaning). Declarative programming is better suited for implementing *deductive systems*, and would benefit from studies of parsing in this context.

In this paper we reconsider a class of context free grammars that are closely related to LR grammars and are likewise capable of describing all deterministic context-free languages. These grammars exhibit characteristics that can be used to formulate a parsing strategy in the framework of logical inference. The inference rules for these parsers are specialized versions of the two basic "shift and reduce" rules above. However, they 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.

2 Bounded Right Context and LR Grammars

The type of grammar we consider suitable for formulating deterministic parsing *as deduction* are known as *bounded right context* (BRC) grammars, introduced by Floyd [4]. The principal characteristic of a BRC grammar is that the unique "handle" of a bottom-up, rightmost 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 look back at the *entire* stack. Implementations of LR parsers rely on deterministic finite state machines to keep

[1] The representation of parsers in the style of logical inference rules was introduced in [14].

track of stack contents. *All BRC grammars are also LR grammars and all LR grammars have BRC equivalents that recognize the same language* (see [10, 7]). The essential difference between BRC grammars and LR grammars that are not already BRC can be illustrated by the following $LR(0)$ grammar:

$$
\begin{array}{l}
S \to aA \mid bB \\
A \to (A) \mid x \\
B \to (B) \mid x
\end{array}
$$

A "reduced-reduce" conflict would exist between $A \to x$ and $B \to x$ unless one keeps track of whether an 'a' or a 'b' was the first symbol read. This is the critical information maintained by the $LR(0)$ state machine of this grammar. An equivalent BRC grammar, preserving the distinction between A and B, can be given as follows:

$$
\begin{array}{l}
S \to aA \mid bB \\
A \to (A) \mid x \\
B \to PB) \mid x \\
P \to (
\end{array}
$$

That is, by *looking back* one symbol to the left, one can now determine the same "state" information, which is now carried by the non-terminal symbol P. State machine generation for this grammar can be avoided.

Because BRC grammars are subsumed by the larger LR class, their development as a tool for parser specification was appearantly halted. It is not our intention here to compete with other parsing algorithms, except we note that most implementations of LR parsing are also restricted to subclasses (SLR and LALR). Our reason for resurrecting the BRC subclass is that the trade-off they offer compared to general LR grammars is a positive one *in the context of logic programming.* Due to the lack of arrays, pointers and mutable variables, the computation of state information necessary for efficient LR parsing is precisely the kind of programming that many current declarative languages are not best suited for. Logic programming is better suited for the specification of *deductive* systems. The simplification afforded by BRC grammars can be used to formulate deterministic parsing as such a system, one that can take advantage of the declarative syntax and unification capabilities of logic programming.

An indication of the practical suitability of BRC grammars for use in compiling is that every LR grammar that appears in the most popular compiler texts (including [1] and [2]) are in fact also BRC grammars. When required, the modifications needed to form BRC grammars from LR grammars are usually few and similar to that of the above example (a more practical example is given in Section 3.2). General algorithms for translating LR grammars into BRC equivalents also exist (see [6]). We shall also introduce a simplification of BRC grammars that also suffices for most parsing needs in Section 5.1.

3 Formal Definitions

The following technical presentation 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 $\overset{*}{\Longrightarrow}_r$ and $\overset{+}{\Longrightarrow}_r$. If $\gamma \overset{*}{\Longrightarrow}_r \epsilon$, the empty sequence, then γ is said to be *nullable*. We also confine our discussion to *reduced* context-free grammars where for all $A \in V$, $S \overset{*}{\Longrightarrow}_r \alpha A \beta \overset{*}{\Longrightarrow}_r w$ 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 $A \overset{+}{\Longrightarrow}_r A$ is possible for any non-terminal A. Reduced grammars allowing such derivations are necessarily ambiguous.

If $S \overset{*}{\Longrightarrow}_r \gamma$ then γ is a *right sentential form* of the grammar. If $S \overset{*}{\Longrightarrow}_r \alpha A w$ 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.

Let $last_l(\gamma)$ be the length-l suffix 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

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

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 that of $LR(k)$ grammars ([10]). Thus it is easy to see why every $BRC(l, k)$ grammar is immediately an $LR(k)$ grammar. Furthermore, BRC grammars are capable of generating the same set of languages as LR grammars, namely all deterministic context-free languages ([10, 7]). In particular, every $LR(k)$ grammar has an equivalent $BRC(1, k)$ grammar and every deterministic language has a $BRC(1, 1)$ grammar. Our formulation below, however, is generalized to $BRC(l, k)$ grammars.

3.1 Valid Handles and Inference Rules

The intended meaning of a shift-reduce *parsing judgement* of the form $\alpha \triangleleft w$, seen in Section 1, is that α is a viable prefix of right sentential form αw of the grammar under consideration. α will be used to represent a sequence of grammar symbols and w a sequence of terminal grammar symbols.

The symmetrical nature of BRC grammars suggests that we augment each grammar with an implicit production $S' \rightarrow \$_0 S \$_1$, 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 $(\beta_l, A \rightarrow \gamma, a_k)$ such that $A \rightarrow \gamma$ 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 $S' \rightarrow \$_0 S \$_1$,

1. $(\$_0, S \rightarrow \gamma, \$_1)$ is a valid handle for each production $S \rightarrow \gamma$.
2. if $(\beta_l, A \rightarrow \sigma B \gamma, a_k)$ is a valid handle, $\beta'_l = last_l(\beta_l \sigma)$, and $a'_k \in first_k(\gamma a_k)$, then $(\beta'_l, B \rightarrow \rho, a'_k)$ is a valid handle for each production $B \rightarrow \rho$.

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. $(\beta_l, A \rightarrow \gamma, a_k)$ *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 $(\$_0, S \rightarrow \gamma, \$_1)$ and follow the above inductive definition until a closure is formed (the computation is also bounded by l, k and the number of productions and 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:

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

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

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

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[2]

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

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 simplifies to $a \in first(G_{i+1})$.

The $first$ relation can be pre-computed so it is not necessarily a costly condition to verify.(and if all shift rules are collapsed into one default rule in actual parser code, the $first$ condition becomes unnecessary).

Determinicity of the canonical inference rules can be checked by pairwise unification of their bottom sides, which should share no common instance. Formally, A *reduce-reduce* conflict between two distinct inference rules exists if they are of the forms

$$\frac{\alpha\beta_l A \lhd a_k w}{\alpha\beta_l\gamma \lhd a_k w}\ Reduce \quad \text{and} \quad \frac{\alpha'\beta'_l A' \lhd a_k w'}{\alpha'\beta'_l\gamma' \lhd a_k w'}\ Reduce$$

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

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

$$\frac{\alpha\beta_l A \lhd a a'_{k-1} w}{\alpha\beta_l\gamma \lhd a a'_{k-1} w}\ Reduce \quad \text{and} \quad \frac{\alpha'\beta'_l\gamma' a \lhd a'_{k-1} w'}{\alpha'\beta'_l\gamma' \lhd a a'_{k-1} w'}\ Shift$$

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 exist no shift-reduce or reduce-reduce conflicts.

[2] It can be shown that G_{i+1} is nullable only if $G_{i+1} \overset{*}{\Longrightarrow}_r B\sigma$ 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 $B \to \epsilon$.

3.2 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:

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

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

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

These in turn give rise to the following deterministic inference rules

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

Without the lookback symbol $\$_0$, there would exist a reduce-reduce conflict between the productions $A \to Aa$ and $A \to a$. 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 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:

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

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

$$\frac{\alpha aA \lhd dw}{\alpha ac \lhd dw}\ R \qquad \frac{\alpha aB \lhd ew}{\alpha ac \lhd ew}\ R \qquad \frac{\alpha bA \lhd ew}{\alpha bc \lhd ew}\ R \qquad \frac{\alpha bB \lhd dw}{\alpha bc \lhd dw}\ R$$

The final example shows where in the syntax of a programming language do BRC grammars differ from general LR grammars. Consider a language with function *definitions* of the form `function f(x) = ...` and function *calls* of the form `f(a)`. A possible source of conflict is that in the definition header `x` can only be an individual identifier, whereas in a function call `a` can be an arbitrary expression. A possible LR grammar for this syntax would be

$F \to function\ id(id) = E$
$E \to id \mid E + T \mid (E) \mid \ldots$

Without the $function$ keyword, a shift-reduce conflict would result when reading an identifier (id) inside parentheses. A BRC grammar would require the following modification:

$F \rightarrow Gid) = E$
$G \rightarrow function\ id($
$E \rightarrow id \mid E + T \mid (E) \mid \ldots$

The unique non-terminal symbol G allows inference rules to distinguish the appropriate context using a single lookback:

$$\frac{\alpha Gid) \triangleleft w}{\alpha Gid \triangleleft)w}\ Shift \qquad \frac{\alpha(E \triangleleft)w}{\alpha(id \triangleleft)w}\ Reduce$$

This modification is essentially the same as for the example in Section 2, and all required modifications we have encountered are of this nature. Only two modifications of this type were required in defining a BRC grammar for an experimental imperative language. Occassional grammar modifications, including such seemingly redundant productions, are sometimes also needed for the SLR and LALR simplifcations of LR parsing. Users of any parser generator must be aware of the requirements of the underlining grammar formalism.

4 Correctness

The formal correctness of our deductive parsing mechanism consists of lemma 2 plus the following results. We use "deduction" to mean the application of a sequence 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 $S' \stackrel{*}{\Longrightarrow}_r \alpha A w$, 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, which is 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 $\alpha_2\beta_2 w_2$[3]. Once we know that $\alpha_2\beta_2 = \sigma\beta$, $A \to \beta = A_2 \to \beta_2$ 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 $A \overset{+}{\Longrightarrow}_r A$ is not possible). However, without any lookbacks or lookaheads, any grammar with an ϵ-production has infinite deductions in reverse. The general termination result is state thus:

Theorem 7. *There are no infinite 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 $(b, A \to \gamma, a)$ and $(b, A \to \gamma, a')$ 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 have the form $(b, A \to \gamma, \{a, a'\})$. Handles of the form $(b', A \to \gamma, a)$ and $(b, A \to \gamma, a)$ can similarly be merged safely. Handles of the form $(b, A \to \gamma, a)$ and $(b', A \to \gamma, a')$, however, can not in general be merged without causing new conflicts (consider the second example of Section 3.2). In the representation of valid handles we can therefore use sets for both the lookbacks and lookaheads. Handle merging is an important implementation-wise feature because it significantly reduces the number of inference rules needed.

For example, consider the following $BRC(1,1)$ grammar:

$S' \to \$_0 S \$_1$
$S \to L = R \mid R$
$L \to *R \mid id$
$R \to L$

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

– $(\{\$_0\}, S \to L = R, \{\$_1\})$, $(\{\$_0\}, S \to R, \{\$_1\})$

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

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

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

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

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 finite automaton of a $LR(1)$ parser[4]. 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:

$S' \to \$_0 E \$_1$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

The fully merged valid handles of this grammar are:

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

The inference rules derived from these handles remain deterministic. Note in particular the absence of $+$ among the lookbacks of the handles for $E \to T$, and the absence of $*$ from the lookbacks for $T \to F$. 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

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

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[5]. 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 *λProlog* (*Teyjus* implementation [11]). 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   % attributed context free grammar for online calculator
[
  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)
].
```

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 infix symbol `==>` seperating the non-terminal from a list of right hand side symbols. The last component of a `rule` term is a semantic action in the form of a λProlog goal (which takes the place of C code in Yacc). Unlike Yacc-style generators, a separate parser is not needed for the grammar specification.

It is required that no attribute variable in a list of productions appear more than once, except in the semantic action goals. Also, only variables can appear as attribute arguments in a production. Without such restrictions, the grammar may become context *sensitive* (DCGs have the same problem).

[5] 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.

Valid handles are generated by forming a closure of triples following the inductive definition as described in the foregoing. The *first* relation needed for lookaheads is pre-computed and written to the parser file as atomic clauses (for efficient lookup). 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.

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[6]:

```
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.
```

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

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 rule, which is used only if no reduce rule is applicable. 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 maximum number of terminal symbols on the right-hand side of any grammar production (which limits the number of symbols that can be shifted before a reduce rule must be applied).

7 Related Work

In Section 1 we have already discussed the relationship between our approach and some other works on parsing in logic programming. 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. 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[7]. 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].

[7] 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].

8 Conclusion

When faced with the need for a generic parsing tool appropriate for use in compiler writing, the logic programmer can consider several alternatives. DCGs are general, but non-deterministic. Using an existing parser generator in an alien language is another alternative. It is technically possible to extract the parse table from code generated by Yacc and use it in a Prolog parser. However, it is difficult to see how such a process can be automated, especially when declarative semantic attributes must be attached to grammar symbols. A forced implementation of established LR parsing algorithms is also possible, and would at least allow semantic actions to be defined in the native language. If such a tool already exists for the declarative language in question, then it may seem needless to examine another grammar formalism. However, when faced with the task of developing a parser generator from scratch, it is valid to address the fact that certain aspects of general LR parsing renders its formulation as declarative, logical clauses highly awkward. Aspects of LR parsing that give it its efficiency also may not translate into anything meaningful in a logic programming context. Perhaps future logic programming languages, such as those based on linear logic, can give declarative formulations of the kind of computation required by general LR parsing. But the past also deserves some consideration. The almost-forgotten class of BRC grammars were never considered in the light of logic programming. BRC grammars, and our SBRC grammars, offer simplifications of LR parsing that are conducive to logic programming, just as LALR and SLR grammars are appropriate simplifications for conventional languages.

Our parser generator has been used in the implementation of an experimental imperative language (with static scoping, functions and types). A parser for most of λProlog 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 "object-logics" into λProlog host syntax. Currently, it is being used to develop a "declarative scripting language" that extends HTML[8].

This paper has intentionally not addressed the detailed representation of semantic attributes and actions. We have completely separated the basic parsing mechanism so that it can be incorporated into a variety of declarative settings. However, it was the desire to reason with a new class of abstract syntax, namely higher order abstract syntax, that truely motivated this work. It is hoped that the availability of this parsing tool will facilitate the exploration of a host of possible applications of declarative programming in general, and of higher order logic programming in particular.

For future work on the parser generator we plan to improve its performance and usability, especially in the form of additional `"error"` clauses for error correction.

[8] Please see accompanying homepage (http://www2.trincoll.edu/~cliang/parsergen/) for more details on these applications.

Acknowledgments

Support for this work was provided by Dale Miller at the Pennsylvania State University, whom the author thank for helpful advice and comments. The author also wishes to thank Jeremie Wajs for testing the parsers.

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 bounded 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. R. Ochitani K. Uehara and O. Kakusho. A bottom-up parser based on predicate logic. *IEEE Proceedings, International Symposium on Logic Programming*, pages 220–227, 1984.
10. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
11. 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.
12. U. Nilsson. AID: an alternative implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.
13. F. Pereira and D. Warren. Definite clause grammars for language analysis. *Artificial Intelligence*, 13:231–278, 1980.
14. F. Pereira and D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.
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.

This article was processed using the LaTeX macro package with LLNCS style