Generalised Recursive Descent Parsing and Follow-Determinism

Adrian Johnstone and Elizabeth Scott*

Royal Holloway, University of London

Abstract. This paper presents a construct for mapping arbitrary non-left recursive context-free grammars into recursive descent parsers that: handle ambiguous grammars correctly; perform with LL(1) efficiency on LL(1) grammars; allow straightforward implementation of both inherited and synthesized attributes; and allow semantic actions to be added at any point in the grammar. We describe both the basic algorithm and a tool, GRDP, which generates parsers which use this technique. Modifications of the basic algorithm to improve efficiency lead to a discussion of *follow-determinism*, a fundamental property that gives insights into the behaviour of both LL and LR parsers.

1 Introduction

Practical parsers for computer languages need to display near-linear parse times because we routinely expect compilers and interpreters to process inputs that are many thousands of language tokens long. Presently available practical (near-linear) parsing methods impose restrictions on their input grammars which have led to a 'tuning' of high level language syntax to the available efficient parsing algorithms, particularly LR bottom-up parsing and, for the Pascal family languages, to LL(1) parsers. The key observation was that, although left to themselves humans would use notations that were very difficult to parse, there exist notations that could be parsed in time proportional to the length of the string being parsed whilst still being easily comprehended by programmers.

It is our view that this process has gone a little too far, and the feedback from theory into engineering practice has become a constraint on language design, and particularly on the design of prototype and production language parsers.

When building a new language, the natural design process starts with the language itself, not with the grammar, and even expert users of existing linear-time parsers are unlikely to produce conflict free LL(1) or LR grammars at their first attempt. In fact grammar debugging is *hard*. Even when the designer has produced a grammar that is acceptable with no (or few) conflicts, it is usually the case that, as a result of logical errors in the design of the grammar, the generated parser either parses some strings that are not in the language or fails to parse some that are. The analogy with program design is strong. In order to effectively design languages we need debuggers for parsers which allow the

* Email: `A.Johnstone@rhbnc.ac.uk` or `E.Scott@rhbnc.ac.uk`

user to think in terms of the parser specification (the grammar). Although many languages are developed from existing ones, and so perhaps should be viewed as stepwise refinements of working systems, neophyte (and even experienced) language implementers find it hard to be sure that the language generated by their grammar is precisely the language they have informally sketched. The situation is particularly difficult in the case of bottom up parsers since, even when a grammar has been accepted by the parser generator and successfully tested it may break when semantic actions are added. In addition, bottom up parsers report errors in terms of multiple table entries, and users often find it difficult to relate table entries to entities in the original grammar. This is not to decry the parsing power of the bottom-up approach which, unlike standard top-down techniques, allows left recursion in grammars to be handled in a natural fashion.

2 Overview

In this paper we present a parsing technique called Generalised Recursive Descent (GRD) which can handle all non-left recursive grammars, even ambiguous ones, but which is based on recursive descent, retaining the attractions of this approach. As there is a well-known algorithm [ASU86, pp. 176–178] which allows left recursion to be removed from any grammar (after some preprocessing to remove certain types of grammar rule), in principle our technique can be used for any context-free language. We also present techniques to allow more efficient implementation of GRD. In particular, we analyse a variant of the GRD algorithm which can be used if the input grammars have a property called *follow-determinism*. In the rest of this paper we shall

- present a construct for mapping arbitrary non-left recursive context-free grammars into recursive descent parsers that
  1. handle ambiguous grammars correctly,
  2. perform with LL(1) efficiency on LL(1) grammars,
  3. handle non-left recursive LR(1) grammars with reasonable efficiency,
  4. allow implementation of both inherited and synthesized attributes,
  5. allow semantic actions to be added at any point in the grammar without affecting either the correctness or the efficiency of the parser.
- discuss *follow-determinism*, a fundamental property that gives insights into the behaviour of both LL and LR parsers,
- describe a tool, GRDP, which generates parsers which use this technique.

There are other parser generators which use a generalised form of recursive descent [BB95, PQ96]. Such parsers usually explore all possibilities at each step in the parse but then select only one to proceed with – a fundamental constraint that causes the parsers to fail even on some non-ambiguous grammars. A common selection procedure is to choose the so-called 'longest-match', i.e. the possibility which matches the longest section of the input string. This process is not guaranteed to produce correct parsers for all grammars. Our parsers return the set of all possible matches at each step and continue with each of them. This allows us to produce correct parsers for all non-left recursive grammars, and all possible derivations in the case where a grammar is ambiguous. The language designer will thus get a correct parser for their grammar which they can then use to test whether their grammar really does generate the language they were hoping for. Once a correct grammar for the language has been developed efficiency issues can be addressed. Our parser generator produces efficient parsers if the designer can refine their grammar so that it has suitable properties. Experimental results for a rstricted form of backtracking recursive descent parsers may be found in [BB95].

We begin by giving a basic definition of formal grammars, to allow us to introduce the terminology and notation that we need.

3 Formal Grammars

We use a slightly unconventional definition of a grammar in which a grammar rule is a mapping from the set of non-terminals to sets of strings of terminals and non-terminals. In the examples in this paper the sets will be finite, so grammar rules can be thought of in the usual EBNF way,

$$A\texttt{::=}\{aBb, aCC, \epsilon\} \quad \text{is equivalent to} \quad A\texttt{::=}\ aBb \mid aCC \mid \epsilon.$$

However, much of what we say applies to grammars where the right hand sides of grammar rules can be infinite sets, and in a future paper we shall be discussing such grammars. Thus we use the set based notation here so that this work will not need to be reformulated later.

A *context free grammar* $\Gamma = (U, T, S, P)$ is a set U of symbols, a set $T \subset U$ of terminals, a start symbol $S \in U\backslash T$, and a set P of rules $A\texttt{::=}\tau_A$, where $\tau_A \subseteq U^*$ (the set of strings on U), $A \in U\backslash T$, and there is only one grammar rule for each A. The elements in τ_A are called *alternates* of the rule $A\texttt{::=}\tau_A$.

A *derivation step* is of the form $\alpha A\beta \Rightarrow \alpha\gamma\beta$ where $\alpha, \beta \in U^*$, $A \in U\backslash T$, and $\gamma \in \tau_A$. A *derivation* is a sequence

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k = \beta$$

where $\alpha_{i-1} \Rightarrow \alpha_i$ is a derivation step, $1 \leq i \leq k$. In this case we write $\alpha \stackrel{*}{\Rightarrow} \beta$.

A symbol X is *reachable* if for some $\alpha, \beta \in U^*, S \stackrel{*}{\Rightarrow} \alpha X \beta$.

We define $L(\Gamma)$, the *language generated by* Γ, to be the strings of terminals which can be derived from S. So

$$L(\Gamma) = \{u \in T^* \mid S \stackrel{*}{\Rightarrow} u\}.$$

At a suitable level of abstraction, a parser P is a map from the set of strings in a language to the set $\{success, failure\}$. The idea is that a string is input to P and after a finite amount of time P should terminate and return either $success$ or $failure$. Formally, a grammar Γ *admits* a parser P if

- for all u in $L(\Gamma)$, $P(u) = success$,

– for all u not in $L(\Gamma)$, $P(u) = failure$.

A parser is *conservative* if inputting $u \in T^*$ results in success if $u \in L(\Gamma)$. A conservative parser may result in failure on some strings $u \in L(\Gamma)$.

The goal of a parser is, given an input string, to construct a derivation of that string. Recursive descent parsers use a *top-down left-most* approach; that is, they start with the start symbol and attempt to construct a derivation step-by-step from the left. At each step in the constructed derivation the left-most non-terminal in the string is replaced.

4 Generalised Recursive Descent Parsing

In this section we describe a generalised version of the well-known recursive descent parsing technique. The generalised version does not require the source grammar to be left factored, and, in the case of ambiguous grammars, returns all valid parse trees for the given input string. Our generalised recursive descent (GRD) parsers overcome the need for left factoring by using arbitrary amounts of backtracking, and they handle both local and global ambiguity by returning sets of sentential forms at each stage rather than selecting a single sentential form for further processing.

One of our reasons for preferring top down parsing is that, as Terence Parr noted in a recent SIGPLAN notices article, 'parsing is not translation'[PQ96]. The significance of this comment is that bottom up parsers are fragile with respect to semantic action placement. In a shift-reduce parser, pending matches are kept on a stack. By definition, shift operations are associated with input strings whose grammar production has not yet been completely identified and so semantic actions cannot be associated with shift operations. Once a production has been completely matched, a reduction operation occurs at which point a semantic action may be executed.

To the parser generator user, the effect of this constraint is that in order to execute a semantic action mid-way through a production that production must be split into a prefix production and a suffix production, with the action being placed at the end of the prefix. Unfortunately there is no guarantee that the resulting grammar will still be accepted by the parser generator. In the worst case, an LALR grammar with a semantic action to be executed after each terminal will be reduced to the same parsing strength as an LL(1) grammar [Par93].

GRD parsers display all of the attractive features of traditional LL(1) recursive descent parsers in that semantic actions may be placed anywhere within the grammar without perturbing the behaviour of the parser or the acceptability of the grammar to GRD parser generator; both synthesized and inherited attributes may be implemented simply *via* the implementation language's procedure parameter mechanism and, most importantly of all, there is a very close relationship between the parser code and the equivalent BNF grammar. This means that a grammar specifying a GRD parser may be debugged by single stepping through the GRD parser using the traditional code debuggers.

Our parsing algorithm and its associated parser generator offer two modes: *prototyping* in which ambiguous grammars are fully supported, even to the extent of returning multiple derivations where no disambiguation predicates are provided, and *production* in which a single derivation is selected on the basis of a property we call follow-determinism. The purpose of the prototyping mode is to allow the designer (and perhaps the theoretician) to explore the properties of ambiguous grammars without restriction. In many cases, the prototyping mode will in practice be fast enough, significantly easing the job of language implementation.

The two parser modes allow a smooth transition from very general parsers to parsers which, whilst still being more general than top-down parsers are competitive in terms of parsing speed. With tongue slightly in cheek, we call our parsers *recursively decent* parsers[2].

Concrete GRD parsers GRD parsers take their input from a read-only buffer with a current character index called `current`. During back-tracking, the index may be moved backwards in the string. We expect, therefore, the whole input to be in memory. This is not a significant constraint in modern systems where memory is plentiful.

GRD parsers contain one parsing function for each grammar rule. On entry to a parser function, the present value of the input index `current` is stored in the local variable `entry_current`. Each parser function constructs a set of indices into the text buffer, corresponding to the position in the text buffer of the `current` index immediately after a sub-string has been recognised. This set is called `return_set`.

The function `mismatch` is used to test the substring at the `current` index against a language token. If the test succeeds, the `current` index is updated to the position in the buffer immediately after the token and FALSE is returned, otherwise the `current` index is left unmodified and TRUE is returned. If a sequence is successfully matched, the position of the current index is added to the `return_set`.

The loops are executed in some arbitrary sequence. (Usually the loops are laid out in the same order as the productions in the grammar to ease the debugging process).

Within each production, each terminal symbol is mapped to a `mismatch` function which compares the portion of the input string starting at the `current` index to the terminal string. A failed match causes the parser to break from the current loop.

Each reference to a non-terminal is mapped to a `for` loop preceded by a call to the corresponding parser function which returns a set of index positions, one for each string matched by that function. These sets can be large, but in an LL(1) grammar, the cardinality of the set is guaranteed to be one or zero. The loop iterates over each element in the set, and the body of the loop comprises the remaining tail of the alternate production.

[2] Thanks to Dan Simpson of Brighton University for this pun.

Pseudo-C code for the GRD parser function corresponding to the production $S::=\{ac, aBc\}$ is shown in Fig. 1.

```
Set S(void)
{
  char *entry_current = current;
  Set return_set = EMPTY_SET;

  if (mismatch("a")) goto branch1_end;
  if (mismatch("c")) goto branch1_end;

  return_set = return_set UNION current;

  branch1_end:

  current = entry_current;

  if (mismatch("a")) goto branch2_end;

  {
    Set String_set = B();

    for (each element in String_set)
    {
      current = current_element_of_String_set;
      if (mismatch("c")) continue;
      return_set = return_set UNION current;
    }
  }
  branch2_end:

  return return_set;
}
```

Fig. 1. A concrete GRD function for the production $S::=\{ac, aBc\}$

An example grammar In the rest of this paper we shall use the grammar:

$$A::=\{BA, C, d\} \quad B::=\{abb, ab\} \quad C::=\{c, cd\}$$

GRDP parser trace trees We use a generalisation of the standard derivation tree to display the results of a GRD parse. Our *parser trace trees* are derivation trees augmented with nodes that show the start of each alternate production and the start of each tail test after a non-terminal has been called. Failed nodes are labeled in parentheses. Fig. 2 shows the tree corresponding to the grammar above and the input string *abbc*. The nodes that would be found in a normal derivation tree are shown as rectangles, and our augmented nodes as ellipses. The 'continuation' nodes that occur after each non-terminal in a sequence are labeled with $\ldots u$ where u is the remainder of the sequence to be parsed. In the case of a parse that returned a single derivation, displaying only the rectangular nodes along branches with successful continuations produces a standard derivation tree.

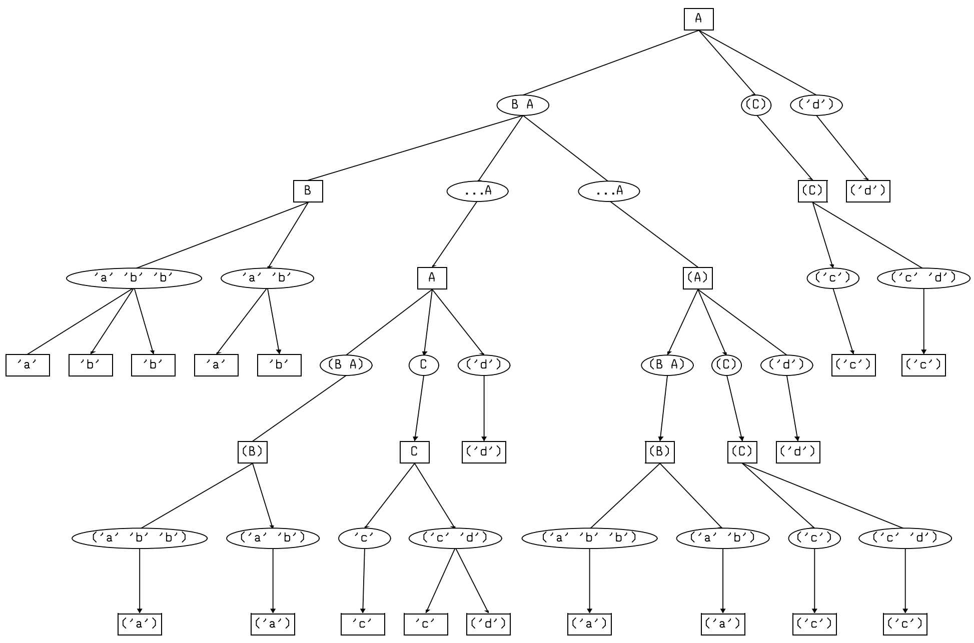

Fig. 2. GRD parser trace tree: example grammar (prototype mode) on string *abbc*

5 Pruning the Search Space

We now consider two ways of improving the efficiency of GRD parsers. The first is to only explore alternates with appropriate 'first' sets: this does not impose any additional restrictions on the source grammar. The second uses lookahead to decide whether to terminate a function call, and to be guaranteed correct it requires the source grammar to possess a property called *follow-determinism* which we describe below.

FIRST **set checking** The version of GRD described so far handles ambiguous grammars by exhaustive searching of the grammar rules. The only concession to efficiency is that the search of an alternate is aborted as soon as a terminal mismatch is discovered. However, there is no point exploring the result of replacing a non-terminal by a particular alternate if that alternate cannot generate either a string beginning with the current input symbol or ϵ, the empty string.

Consider the example grammar in the previous section. A prototype mode GRD parser for this grammar will produce the trace shown in Fig. 2 on input string *abbc*.

We can improve the efficiency of the prototyping parser, without imposing any further restriction on the source grammar, by prefacing each alternate and non-terminal call with a test to ensure that `current` is pointing to a substring that is in the FIRST set for that alternate or non-terminal.

Formally FIRST sets are defined as follows:

$$\text{FIRST}(\alpha) = \begin{cases} \{a \in T \mid \alpha \stackrel{*}{\Rightarrow} av\} \cup \{\epsilon\}, & \text{if } \alpha \stackrel{*}{\Rightarrow} \epsilon \\ \{a \in T \mid \alpha \stackrel{*}{\Rightarrow} av\}, & \text{otherwise.} \end{cases}$$

With the FIRST set test, a GRD parser for the above grammar will produce the trace shown in Fig. 3 on input *abbc*

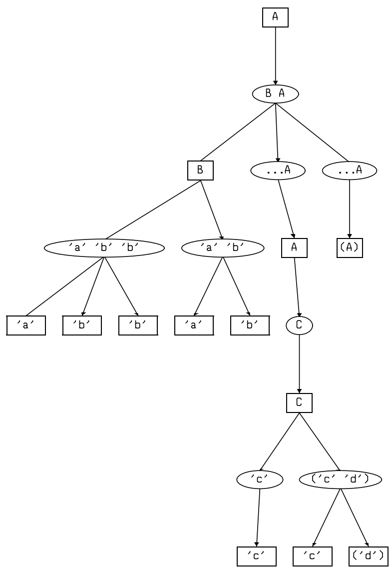

Fig. 3. Pruning the search tree with FIRST sets

The effect of adding such tests is to lop off entire branches of the parser trace tree, which significantly aids efficiency. More particularly, if the user is able to left factor their grammar, then at most one branch will be entered within each production. A future version of the parser generator will generate profile information from running parsers on a large and representative set of sample strings and calculating the probability of each branch being entered on typical input strings. By ordering the branches in descending order of probability, further speedups will be obtained in the production version of GRD.

FOLLOW set checking In practice, we expect users to switch from the prototype mode to *production* mode at some stage in the development of their language. In production mode each parser function returns at most one string. We select the string on the basis of the next input symbol, relying on a property called *follow-determinism*. This modification provides similar parsing power to the traditional longest-match approach, and is guaranteed to produce a correct parser for a follow-determined grammar. In addition, follow-determinism is generally more efficient than longest-match because we can abort an alternate as soon as we find a follow-determined match, whereas longest match requires all strings to be checked and compared for length. Follow-determinism is discussed in detail in [JS97b]. Here we just give an overview of the basic ideas.

Suppose that a GRD parser for the grammar

$$A\texttt{::=}\{BA, C, d\} \quad B\texttt{::=}\{abb, ab\} \quad C\texttt{::=}\{c, cd\}$$

has been given input $abbc$ and that so far it has constructed the step:

$$A \Rightarrow BA$$

On exploring the replacement of B for the next step in the derivation a parser will be able to match both alternates of the rule.

A parser generated by the prototyping version of GRD will simply return both matches and continue to develop both corresponding derivations.

A longest-match based parser will find all matches but will select the longest and only pursue that corresponding derivation.

When a production GRD parser finds a match it checks to see whether the current input symbol is in the FOLLOW set (see below) of the current non-terminal. If so it just selects that match and doesn't explore any further alternates. So in the above example, as c is in the FOLLOW of B, the next step constructed would be

$$A \Rightarrow BA \Rightarrow abbA$$

and the alternate ab would not even be explored. Eventually the trace tree shown in Fig. 4 would be produced.

Follow-determined grammars In the case where there is a choice of matches which can be used in the next step of a derivation, selecting one to proceed with may cause a problem. It may turn out to be impossible to complete the derivation with that choice, while a different choice would have resulted in success. This is a general problem for techniques which select one of a choice of matches. There are grammars for which choosing the longest match results in failure, whereas

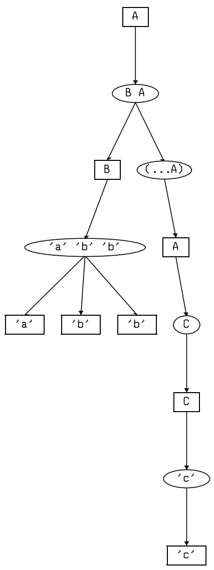

Fig. 4. Pruning the search tree with FOLLOW sets

choosing a shorter match would have allowed the successful completion of the derivation. However, it is possible to define the property required of a grammar to ensure that selecting the follow-determined match will always result in a complete derivation, if one exists.

The FOLLOW set of A is the set of terminals which can appear immediately after A in a string derived from the start symbol, together with a special end of file symbol \$ if A can appear at the end of a such a string. So

$$\text{FOLLOW}(A) = \begin{cases} \{a \mid \text{for some } \alpha, \beta,\ S \stackrel{*}{\Rightarrow} \alpha A a \beta\} \cup \{\$\}, & \text{if } S \stackrel{*}{\Rightarrow} \alpha A \\ \{a \mid \text{for some } \alpha, \beta,\ S \stackrel{*}{\Rightarrow} \alpha A a \beta\}, & \text{otherwise,} \end{cases}$$

where S is the start symbol of the grammar.

Definition A grammar Γ is *follow-determined* if for every $A \in U$, if $A \stackrel{*}{\Rightarrow} u \in T^*$ and $A \stackrel{*}{\Rightarrow} uaw$, for some $a \in T$, $w \in T^*$, then $a \notin \text{FOLLOW}(A)$.

If the source grammar is not left recursive then a production version GRD parser for it will be conservative. If the source grammar is also follow-determined it will admit a production version GRD parser. Note: a follow-determined grammar will also admit a longest-match based parser, but as noted above, longest match will be less efficient in general.

A non-LR(1), non-follow-determined grammar which does admit a production-mode GRD parser We now look at an (ambiguous) non-follow-determined grammar which, none-the-less, admits a production version GRD parser, without the need for special ambiguity breaking measures.

$$S::=\{BAc\} \qquad A::=\{a,\ aA\} \qquad B::=\{b,\ ba\}$$

We have $B\overset{*}{\Rightarrow}b$, $B\overset{*}{\Rightarrow}ba$, and $S\overset{*}{\Rightarrow}Bac$ so the grammar is not follow-determined. The corresponding language is $\{ba^nc \mid n \geq 1\}$. When parsing ba^nc, $n \geq 2$, the non-terminal B may generate b or ba, but in either case, the non-terminal A will generate then remaining string of a's and the parse will succeed.

It is worth noting that a longest-match parser will fail on the string bac.

6 Some Properties of Follow-Determined Grammars

As we have already discussed, there are two classes of GRD parser: the prototyping versions which are admitted by any non-left recursive grammar, and the more efficient production versions which are only guaranteed to be admitted by non-left recursive follow-determined grammars. We shall now describe some theoretical aspects of follow-determined grammars and the languages which can be specified with them. Because of space constraints the proofs of almost all the results quoted have been omitted, but complete proofs can be found in [JS97b].

6.1 LL(1) Grammars

We begin by looking at the relationship between LL(1) grammars and follow-determinism. First we note that currently used recursive descent based parsers require the grammar to be free of left recursion, as left recursion can cause the parser to go in to an infinite loop.

A grammar Γ is *left recursive* if for some $A \in U$ and $\alpha \in U^*$, $A\overset{*}{\Rightarrow}A\alpha$, where the derivation has at least one step.

A grammar Γ is *left factored* if for every non-terminal $A \in U$ and for every pair $\alpha, \beta \in \tau_A$ with $\alpha \neq \beta$, we have $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$. Left factoring avoids the need to explore more than one alternate at each derivation step.

The other property required of an LL(1) grammar is that the parser should know 'when to stop' matching a rule. This may seem a little strange, but essentially we mean that if the parser has matched a portion of the input to the right hand side of a rule, it must be clear whether it should continue to try to get a longer match, or stop and begin matching the next rule. The grammar

$$S::=\{Aa\} \qquad A::=\{aA,\ \epsilon\}$$

has the disjoint FIRST property above but is not LL(1). The problem illustrated by this grammar is the basis of the follow-determinism constraint.

Follow-determinism is not the condition usually used in the definition of LL(1) grammars, but it turns out that the weaker condition usually stated is equivalent to follow-determinism in the presence of left factoring.

Γ is *simply follow-determined* if for all A such that $A\overset{*}{\Rightarrow}\epsilon$, if $\beta \in \tau_A$ then $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$.

A grammar Γ is said to be LL(1) if $L(\Gamma) \neq \emptyset$, Γ is left factored and Γ is simply follow-determined. Theorem 1 (below) shows that all LL(1) grammars are follow-determined. The following lemma is used in the proof of Theorem 1.

Lemma 1. *Let Γ be an LL(1) grammar. If $S \stackrel{*}{\Rightarrow} \alpha_1 \alpha \alpha_2 \in U^*$ and if for some $u \in T^*$, $a \in T$ and $\gamma \in U^*$ we have $\alpha \stackrel{*}{\Rightarrow} u$, $\alpha \stackrel{*}{\Rightarrow} ua\gamma$ then there exists a non-terminal A such that $\alpha = \alpha' A \omega$, where $\omega \stackrel{*}{\Rightarrow} \epsilon$ and $a \notin$ FOLLOW(A).*

Theorem 1 *1. If Γ is follow-determined then Γ is simply follow-determined.*
2. If Γ is left factored and simply follow-determined then Γ is follow-determined.

Proof. (1) Suppose that Γ is not simply follow-determined. Then there exists $A \in U$ and $\beta \in \tau_A$ such that

$$A \stackrel{*}{\Rightarrow} \epsilon, \qquad A \Rightarrow \beta \stackrel{*}{\Rightarrow} a\beta'$$

for some $a \in$ FOLLOW(A). But then taking $u = \epsilon$ we have $A \stackrel{*}{\Rightarrow} u$, $A \stackrel{*}{\Rightarrow} ua\beta'$, so Γ is not follow-determined.

(2) Suppose that for some reachable A and some $u \in T^*$, $a \in T$, $\gamma \in U^*$ we have $A \stackrel{*}{\Rightarrow} u$ and $A \stackrel{*}{\Rightarrow} ua\gamma$. Then by Lemma 1 we have $a \notin$ FOLLOW(A), as required.

Corollary 1 *LL(1) grammars are left factored and follow-determined.*

6.2 Follow-Determinism and Left Factoring

Recursive descent based parsers are more efficient on grammars which are left factored. However, we shall now show that there exist follow-determined grammars for which there are no equivalent left factored grammars, thus the capability of production version GRD parsers to deal with non-left factored grammars increases the number of languages which can be handled.

Let $\Gamma = (U, T, S, \mathcal{P})$ be a context free grammar. For $a \in T$ and $\alpha \in U^*$ define $L_a(\alpha)$ to be the set of strings of terminals beginning with a which are derivable from α. So

$$L_a(\alpha) = \{av \mid \text{for some } v \in T^*,\ \alpha \stackrel{*}{\Rightarrow} av\}.$$

The idea is to show that if there is a non-terminal B in Γ such that

$$L_a(B) = \{aba, a^2b^2, a^3b^3a, a^4b^4, \ldots\} = L, \text{ say}$$

then Γ cannot be left factored. We then demonstrate that this language L can be generated by a follow-determined context free grammar, and since it must be $L_a(S)$ for the start symbol of any grammar which generates it, L cannot be generated by a left factored grammar.

The main part of the proof relies on the following lemma.

Lemma 2. *Suppose that $\Gamma = (U, T, S, \mathcal{P})$ is a left factored context-free grammar.*

1. *If there exists $B \in U \backslash T$, $a, b \in T$, and $J \geq 0$ such that*

$$L_a(B) = \{ab^{J+1}a, a^2b^{J+2}, a^3b^{J+3}a, a^4b^{J+4}, \ldots\} = X_J, \text{ say,}$$

then there exists $C \in U$ such that

$$L_a(C) = \{ab^{J+2}, a^2b^{J+3}a, a^3b^{J+4}, a^4b^{J+5}a, \ldots\} = Y_{J+1}, \text{ say.}$$

2. If there exists $B \in U \backslash T$, $a, b \in T$, and $J \geq 0$ such that

$$L_a(B) = \{ab^{J+1}, a^2b^{J+2}a, a^3b^{J+3}, a^4b^{J+4}a, \ldots\} = Y_J, \text{ say,}$$

then there exists $C \in U$ such that

$$L_a(C) = \{ab^{J+2}a, a^2b^{J+3}, a^3b^{J+4}a, a^4b^{J+5}, \ldots\} = X_{J+1}, \text{ say.}$$

Theorem 2 *There exists a non-left recursive, follow-determined grammar for which there is no corresponding left factored context free grammar.*

Proof. Let $L = \{a^{2n-1}b^{2n-1}a, a^{2n}b^{2n} \mid n \geq 1\}$.

$$S::=\{aAba,\ aBb\} \qquad A::=\{aBb,\ \epsilon\} \qquad B::=\{aAb\}$$

is a follow-determined grammar for L which is not left recursive.

Suppose that $\Gamma = (U, T, S, \mathcal{P})$ is a left factored context free grammar which generates L. Then clearly,

$$L_a(S) = \{aba, a^2b^2, a^3b^3a, a^4b^4, \ldots\} = X_0.$$

So, by Lemma 2(1), there exists a non-terminal C_1 such that

$$L_a(C_1) = \{ab^2, a^2b^3a, a^3b^4, \ldots\} = Y_1.$$

Suppose that there are non-terminals $C_1, \ldots, C_{2n-1}$ such that

$$L_a(C_i) = \begin{cases} Y_j, & \text{if } i = 2j-1 \\ X_j, & \text{if } i = 2j. \end{cases}$$

Then, by Lemma 2(2) there exists a non-terminal C_{2n} such that $L_a(C_{2n}) = X_n$. Then, by Lemma 2(1) there exists a non-terminal C_{2n+1} such that $L_a(C_{2n+1}) = Y_{n+1}$.

Since the X_i and Y_i are all different we must have $C_r \neq C_s$ if $r \neq s$ and thus Γ has infinitely many distinct non-terminals. So Γ cannot be a context free grammar.

6.3 Follow-Determinism and LR(1) Grammars

The ability of LR parsers (see [ASU86]) to accept left recursive grammars means that they can be used with some grammars which do not admit generalised recursive descent techniques unless they have been preprocessed by the left recursion removal algorithm. One of our reasons for preferring GRD parsers is that they provide helpful error diagnostics. (LR parser generators usually detect grammar problems by finding multiple entries in a parse table, but the user wants to know what the problem is in terms of the grammar rules, not in terms of the parse table, so that the grammar can be modified to remove the problem.) However, it is also the case that there are GRD parsers for some grammars whose languages cannot admit an LR parser. This is clear in the case of the prototyping version of GRD, because it can handle ambiguous grammars. The fact that the production version of GRD can be used on some non-LR languages is shown by the following theorem.

Theorem 3 *There is a language L which has a follow-determined context-free grammar but which is not LR. In particular there is no LR(1) grammar for L.*

Proof. Let
$$L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}.$$
It is known, see [AU72] that L cannot be generated by an LR grammar.

Clearly the grammar

$$S::=\{aAb, aBb^2\} \quad A::=\{aAb, \epsilon\} \quad B::=\{aBb^2, \epsilon\}$$

generates L. If Γ is not follow-determined then we must have

$$A \stackrel{*}{\Rightarrow} \alpha u, \quad A \stackrel{*}{\Rightarrow} \alpha ubv \qquad \text{or} \qquad B \stackrel{*}{\Rightarrow} \alpha u, \quad B \stackrel{*}{\Rightarrow} \alpha ubv$$

for some $\alpha \in U^*$ and $u, v \in T^*$. But

$$A \stackrel{*}{\Rightarrow} a^n A b^n \text{ or } A \stackrel{*}{\Rightarrow} a^n b^n$$

so we cannot have
$$A \stackrel{*}{\Rightarrow} \alpha u, \qquad A \stackrel{*}{\Rightarrow} \alpha ubv.$$
Similarly, we cannot have

$$B \stackrel{*}{\Rightarrow} \alpha u, \qquad B \stackrel{*}{\Rightarrow} \alpha ubv.$$

Thus Γ is follow-determined.

7 The GRDP Parser Generator

GRDP is a tool which reads a BNF language specification and outputs an ANSI-C program which implements a generalised recursive descent based parser for that language. The GRDP source syntax is based on, and backwards compatible with our existing RDP LL(1) parser generator[Joh95]. As well as the BNF rules, the source file may contain declarations to control scanner behaviour and to switch GRDP into production mode. By default GRDP generated parsers use the prototyping mode of the GRD algorithm in which sets of strings are returned by parser functions. These parsers correctly handle even ambiguous grammars, returning multiple derivations. In production mode, parser functions return at most one string, being the first follow-determined match discovered in the rule.

GRDP is written using RDP and makes use of RDP's integrated symbol table handling, set manipulation and graph drawing libraries. An option within GRDP enables the construction of parser trace trees such as those shown in this paper. These trees are output in a form suitable for display with the VCG graph visualisation tool[San95] and form an extremely useful debugging aid.

The present version of GRDP is limited to traditional BNF only. Our approach to extended BNF structures, such as Kleene closure relies on our permutation iterator construct which subsumes the optional phrase, Kleene closure and positive closure operators found in some EBNF variants. We shall present a fuller version of GRDP in a future paper, in association with the theory of permutation iterators. GRDP, like its predecessor RDP, is fully public domain. Further information may be obtained from the GRDP Web page at `http://www.dcs.rhbnc.ac.uk/research/languages/grdp.shmtl`

8 Conclusions and Further Work

We have presented a construct for mapping arbitrary non-left recursive context free grammars into generalised recursive descent (GRD) parsers that handle ambiguity correctly whilst operating with LL(1) efficiency on LL(1) grammars. We believe that this technique will allow language designers the freedom to write grammars in an unconstrained style that is natural to them whilst developing the syntax and semantics of their target language, and then smoothly address issues of efficiency as part of a refinement process. A wider discussion of GRD and the impact of parser generators on the design of languages may be found in [JS97a]. Our production mode parser relies on the grammar being follow-determined. A theoretical study of follow-determinism including proofs and additional results may be found in [JS97b].

We have implemented a parser generator called GRDP based on these ideas and are using it to investigate the performance of GRD style parsers. We propose to repeat the experiments reported in [BB95] and extend the results to cover GRDP in both modes and also look at ANSI C and ISO-Pascal grammars. We shall extend GRDP to accept EBNF notation and a new 'permutation' operator that allows the specification of free-order constructs as well as subsuming the commonly used EBNF regular expressions.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles techniques and tools.* Addison-Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 (Parsing). Prentice-Hall Inc., 1972.

[BB95] Peter T. Breuer and Jonathan P. Bowen. A PREttier Compiler-Compiler: Generating higher-order parsers in C. *Software Practice and Experience*, 25(11):1263–1297, November 1995.

[Joh95] Adrian Johnstone. RDP – a recursive descent compiler compiler. Technical Report TR-95-10, Royal Holloway, University of London, Computer Science Department, March 1995.

[JS97a] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent. Part 1: language design and parsing. Technical Report TR-97-18, Royal Holloway, University of London, Computer Science Department, October 1997.

[JS97b] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent. Part 2: some underlying theory. Technical Report TR-97-19, Royal Holloway, University of London, Computer Science Department, October 1997.

[Par93] Terence John Parr. *Obtaining Practical Variants of LL(k) and LR(k) for* $k > 1$ *By Splitting the Atomic k-Tuple.* PhD thesis, Purdue University, August 1993.

[PQ96] Terence J. Parr and Russell W. Quong. LL and LR translators need $k > 1$ lookahead. *ACM Sigplan Notices*, 31(2):27–34, February 1996.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.