Generalised recursive descent
Part 1: Language design and parsing

Adrian Johnstone
A.Johnstone@rhbnc.ac.uk

Elizabeth Scott
E.Scott@rhbnc.ac.uk

Technical Report

CSD – TR – 97 – 18

October 9, 1997

Royal Holloway University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

1 Introduction

Engineering is neither science nor mathematics but an independent activity that blends the engineer's intuition with scientifically derived observations and theoretically derived predictions. In this sense, the purpose of theory in engineering is to provide a body of pragmatic mathematics and associated tools that enable the practising engineer to increase their productivity and the reliability and efficiency of the products they design.

One characteristic of a mature engineering subject is the clear feedback of theory into practice: a theory is developed initially to explain observed phenomena and then best practice is defined in terms of the opportunities and constraints predicted by that theory. Computing offers disappointingly few examples of this pattern: many of the core aspects of computing have not yielded satisfactorily to theoretical analysis. It is also unfortunately true that real scientific observations, in the sense of repeatable experiments that systematically investigate the degrees of freedom in a system, are rarely reported in the computing literature.

The impact of context free parsing theory on the design of high level languages is one of the few success stories of computing as engineering. The earliest languages were designed *ad hoc* and are difficult to parse. The difficulties affect both the design of the parser and its execution speed: FORTRAN parsers usually incorporate unpleasant tricks to resolve lexical elements that show context sensitivity and also to handle limited backtracking.

The story of how BNF was used so effectively in the design of Algol-60 is well known [Bac60]. That success led to the widespread study of formal languages and 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 can 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 target language or fails to parse some that are. The analogy with programme design is strong. In order to effectively design languages we need debuggers for parsers which allow the user to think in terms of the parser specification (the grammar). 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.

2 Overview and claims

This report is the first of a two part series in which we address fundamental design issues in the syntax of programming languages and the way in which implementation difficulties have had a deleterious effect on the expresiveness of languages. We

- ◇ present a construct for mapping arbitrary non-left-recursive context-free grammars into generalised recursive descent (GRD) 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 straightforward 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.

We believe that adoption of these techniques will free the language designer to use more natural notations and significantly reduce the compiler-writer's reliance on parser tricks that circumvent the limitations of current parser generators and parsing algorithms.

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.

Our presentation is split into two parts. In this report we give a detailed discussion of the constraints that practical parsing techniques impose on language design. We also describe the new parsing technique, generalised recursive descent, and a tool GRDP which generates a generalised recursive descent parser given a suitable input grammar. In Part 2, [JS97a], we discuss the theoretical aspects of grammars which admit a production version generalised recursive descent parser. In particular, we introduce the concept of *follow-determinism*, and investigate its role in LL(1) and LR grammars.

3 Prototyping and refinement

As a rule, in any engineering project we should like to separate out the specification from the implementation. Of course, there are always limitations: a project specification that implied the exact solution of NP-complete problems would probably be unacceptable even though the customer might see that as an implementation problem showing through. However, we certainly no longer expect programmers to be concerned about the details of the underlying machine they are using. The programmer designs, implements and debugs in a single high level language notation. Slow, but flexible prototypes may be converted to tuned, efficient production programs by a process of transformation, guided by profiling data.

We should like to design languages this way too. When thinking about a new language the natural emphasis is on the sentential forms of the grammar and their associated semantics. Having to simultaneously write the grammar for a language and juggle the sentential forms so as to yield a grammar that is conflict free in the chosen parser technology confuses two fundamentally different design processes.

While designing a grammar, it is convenient to be able to generate a parser to help with the process of checking whether the grammar actually generates the required language. For this process to be effective the parsing technique on which the parser is based should allow easy transfer between parser steps and grammar rules, so that problems identified using the parser can be related directly back to the grammar structure. Top down, recursive descent based techniques are appropriate in this respect. In Section 5 we shall describe a general recursive descent technique which can be used on any non-left-recursive grammar.

If parsers are to be generated for several versions of a grammar, it is necessary to have a tool which can do this. Given BNF, or some related higher level notation for describing languages of strings, we need a tool which will accept meaningful BNF specifications. There are, of course, some meaningless BNF specifications in the sense that it is possible to write BNF productions that never yield any strings, and we would expect these to be treated as syntax-level errors by the tool. In Section 8 we shall describe GRDP, a tool which constructs a (prototyping) parser for any meaningful non-left recursive grammar. In particular, GRDP can generate a parser for an ambiguous grammar; such a parser will yield more than one derivation for some strings.

After the designer has completed the specification of sentential forms and associated semantics, it is time to attack the efficiency of the parser. Our tool comes in two flavours: the prototyping parser generator which will correctly parse all non-left recursive grammars, including ambiguous ones, and the production parser generator which requires each parser function to return a maximum of one string. The production version uses generalised recursive descent together with a test involving a lookahead symbol and a 'FOLLOW' set. It requires the source grammar to be *follow-determined*, a new property discussed briefly below and in detail in [JS97a].

Our aim is to allow the language designer to first design a language and associated grammar concentrating on getting the semantics they require, then refine the grammar so that it admits a reasonably efficient parsing technique. In the future we propose to collect profiling information from running parsers which will be used to back-annotate the grammar with probability measurements. These will be used to guide the parser generator as to the order in which alternate productions of a non-left factored grammar should be investigated.

4 The effect of simple BNF and linear-time parsing on language design

The shape of modern programming languages has been fundamentally constrained by the limitations of standard parsing techniques. For many years the tendency in automated parser generation has been to use 'bottom-up' techniques, and to require that the source grammar be unambiguous. Our tool uses a top-down approach because, as discussed in the previous section, it is important for language design that the parser structure reflects the grammar structure. In this section we look at some areas of language specification that fit badly into the conventional framework but which fit well with generalised recursive descent parsers.

Parsing limitations are not the only factor governing the kinds of construct used by programming language designers. The weak expressive power of BNF means that some useful constructs require very large grammars, and as a result are rarely used. We examine such constructs and some approaches to more expressive BNF style notations in Section 4.5.

4.1 Lookahead

In general, the mainstream programming languages developed since 1965 have a syntax that is designed to be parsable using LL(1) or LR parsers. Pascal is particularly influenced by the constraints of LL(1) languages, although interestingly only at the phrase level: at the lexical level Pascal also requires lookahead. Consider the two language fragments

$\boxed{\texttt{11..22}}$ and $\boxed{\texttt{11.22}}$

The first is an integer range literal describing the sequence (11, 12, ..., 22) whilst the second is a real number literal. One symbol lookahead does not suffice to resolve these constructs: after reading an integer the lexical analyser must look at the next two characters to decide whether it is scanning an integer or a real.

Recently, new programming languages have begun to display un-LALR like developments. For instance, what are we to make of the following C++ phrase?

```
A::B::C::C~C
```

This sequence of scope operators may be arbitrarily long, and therefore requires arbitrary amounts of lookahead to resolve [PQ96]. We might criticise the designers of C++ for introducing such a construct which could surely have been based on a language construct that did not require parser lookahead, but on the other hand this is a completely natural way to write a cascaded scope override so perhaps it is the current engineering practice which is at fault.

It is worth quoting at length from [Str94] for the insights it gives into the processes undergone during language design (as opposed to the implementation of an already worked out syntax). In 1982, Stroustrup was planning the implementation of Cfront, a compiler front end that translated the first C++ dialect into C.

> '...when I first planned Cfront, I wanted to use a recursive descent parser because ...I liked such parsers' ability to produce good error messages and because I liked the idea of having the full power of a general-purpose programming language available when decisions had to be made in the parser. However, being a conscientious young computer scientist, I asked the experts. Al Aho and Steve Johnson were in the Computer Science research Center and they, primarily Steve, convinced me that writing a parser by hand was almost certainly old-fashioned, would be an inefficient use of my time, would almost certainly result in a hard-to-understand and hard-to-maintain parser and would be prone to unsystematic and therefore unreliable error recovery. The right way was to use an LALR(1) parser generator, so I used Al and Steve's YACC[ASU86]
>
> '...In retrospect, for me and C++ it was a bad mistake ...at the time nobody had been able to write an LALR(1) grammar for C. The LALR(1) grammar used for ANSI C was constructed by Tom Penello about a year later. Even Steve Johnson's PCC, which was the preeminent C compiler at the time, cheated at details that were to prove troublesome to C++ parser writers. For example, PCC didn't handle redundant parentheses properly so that
>
> `int (x);`
>
> wasn't accepted as declaration of `x`. Worse, it seems that some people have a natural affinity to some parser strategies...My bias towards top down parsing has shown itself over the years in the form of constructs that are hard to fit into a YACC grammar. To this day, Cfront has a YACC parser supplemented by much lexical trickery relying on recursive descent techniques.'

There can be little doubt that the advice given to Stroustrup was good: writing parsers by hand is a depressing waste of programmers' time performing what is *mostly* a clerical exercise. The problems arose as a result of the choice of parser generator, not from the decision to use a parser generator. In prototyping mode, GRDP is not limited by lookahead at all. In production mode GRDP is like an LR(1) parser generator in that it is not limited by lookahead on entering an alternate but only uses one symbol of lookahead for the follow-determinism.

4.2 Ambiguity

Even Pascal contains a phrase level ambiguity: the 'dangling else' problem. It is interesting in that even in an avowedly LL(1) language like Pascal, the Algol-68 `if ... then ... fi` construct was rejected despite the fact that it produces a cleaner syntax from the point of view of a parser generator.

Any ambiguities that are left in a grammar will usually need to be resolved by the parser. Whilst ambiguous grammars are theoretically interesting, few compiler writers would be pleased if the parser returned more than one derivation tree *per* phrase to be parsed. The usual approach is to use the lexical ordering of the BNF rules to specify disambiguation: this is theoretically uncomfortable because it undermines the declarative nature of a BNF specification: in fact such parsers are really implementing a mathematical notion that we call an *ordered grammar*.

Whilst ambiguous grammars are usually viewed as inherently bad, there is a counter view. More than twenty years ago Aho, Johnson and Ullman[AJ75] wrote that

> Unfortunately, the most natural grammar describing a language is frequently not in the class of grammars for which we are able to construct efficient parsers mechanically...
> ...we suggest using an ambiguous context-free grammar together with disambiguating rules as a method of specifying the syntax of a programming language. We feel that a number of common syntactic constructs in programming languages can be specified more naturally by this technique than by using an equivalent unambiguous grammar. The disambiguating rules are based on familiar concepts from operator precedence and recursive descent parsing...The resulting parsers are often smaller and more efficient than corresponding parsers constructed from unambiguous grammars'

Although undoubtedly attractive, this approach to language design has not in practice become popular, probably because the standard parser generators were never enhanced to allow easy specification of disambiguation constructs, although YACC [Joh75] does encourage the use of precedence directives to resolve ambiguities in operator grammars. It is also true that the examples quoted in [AJ75] (arithmetic expressions and the dangling else problem) have well-known solutions that do not require explicit disambiguation predicates.

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. The purpose of the prototyping mode is to allow the designer (and perhaps the theoretician) to explore the properties of ambiguous grammars without restriction.

4.3 Semantic actions and attributes

As Terence Parr noted in a recent SIGPLAN notices article, 'parsing is not translation' [Par95]. 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 and at this point a semantic action may be executed.

To the parser generator user, the effect of this constraint is that semantic actions may only be placed at the end of a production, or to put it another way if the user wishes to execute a semantic action mid-way through a production then the 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]. This is intuitively obvious: splitting the original productions down will result in productions with only a single terminal, and therefore no shifting can occur. Since the LALR parser derives its power from its ability to delay matching decisions by shifting partially matched phrases onto the stack it is not surprising that an LALR parser that does no shifting is as weak as a one-symbol lookahead predictive parser.

Inherited attribute passing is 'unnatural' in a bottom up parser. Although modifications to the standard bottom-up parser generators are available they are not well integrated into the parser's operation and usually require the user to understand internal details of the parser.

Top-down parsers allow natural placement of semantic actions, recursive descent parsers allow trivial implementation of both inherited and synthesized attributes in terms of parser function parameters. Traditionally the drawback here is that the parser strength is low. Generalised recursive descent has increased parsing strength as it does not require the grammar to be left factored.

4.4 Efficiency

Efficiency considerations have dominated the design of parsing algorithms, and by extension languages. Parser generators in common use today were targeted to 1970's computer systems which were very slow and had very small main memories compared to modern architectures. There was an understandable emphasis on the design of parsing algorithms that allowed strings to be parsed in time proportional to the length of the string. Nevertheless, language designers were not averse to a little judicious lookahead as we have seen. Apart from the raw increase in clock speed, modern machines show two key improvements that affect the choice of parsing algorithm:

1. main memory is now sufficiently plentiful that input buffers much larger than average input text size are cheap, and

2. the overhead of a procedure call is low. In particular, RISC architectures have very efficient procedure linkage compared to, say, the VAX architecture [HP90].

The availability of large memory buffers affects the cost of lookahead: for lookahead implies backtracking and backtracking implies multiple passes over the parser input string. When that string is being read from disk this may impose unacceptable overhead, but if the string is read once from disk, and subsequently from an in-memory buffer then the speed of parsers with limited backtracking is acceptable [BB95].

The procedure overhead affects the cost of recursive descent parsers, which are naturally procedure call intensive but on modern architectures calling a parameterless procedure is not much more expensive than an unconditional jump [MCF+86]. These factors together act to reduce the cost of recursive descent parsing.

4.5 Counting and permuting

The previous two topics (lookahead and ambiguity) concern parsing strength: that is the practical availability of parsers that can cope with whatever the engineer writes in their BNF language specification. A more subtle constraint on language design is the ease (or difficulty) with which some useful constructs may be described using BNF. BNF, and for that matter the EBNF variants based on Wirth's notation [Wir77] and regular expression style notation [ASU86] are very low level mechanisms for describing sets of strings. In particular, the standard variants contain no mechanisms for counting, and permutation of constructs can only be described by enumerating the permutations as individual BNF productions. Since the number of permutations grows factorially, construct permutation is rare in programming languages.

Counting is perhaps most applicable to scanner design: in many languages the length of identifiers is limited to small numbers of characters and in languages such as FORTRAN-IV [McC65] and Occam [Inm88] indentation is significant. At phrase level, batch control languages often limit the number of parameters in a call, and in languages such as C and Algol-68 that allow the definition of multi-dimensional array literals the number of elements in each sub-literal must be counted and matched. The usual approach to such counting problems is to handle the checking with embedded semantic actions, but unfortunately there is no standard notation for describing actions, so grammars are often annotated with English-language comments for publication.

Permutation of constructs is common in languages that are not specified using context free grammars (such as the BibTEX source language [GMS94]) and rare in languages that are specified with context free grammars. The case for extending BNF to support permutation phrases is eloquently put in [Cam93] with examples drawn from C and languages that offer named parameter passing. We find that Cameron's proposed solution is insufficiently powerful, and more importantly breaches the string-based semantics of normal BNF. Cameron's semantics is very prone to generating ambiguous grammars, and cannot handle permuted comma-delimited lists of phrases which is the usual form for keyword based parameter passing. In a later paper we will describe an extension to the parsing algorithm described here that handles permutation strings in a theoretically elegant and practical fashion. The theoretical treatment in this paper is based on BNF phrases that represent finite sets which is sufficient to explain the properties of the generalised parsing algorithm described here. The theory of our permutation operator requires a treatment in terms of infinite sets which is beyond the scope of this paper.

Our GRDP parser generator, described in Section 8 below, has been designed to allow extension to permutation grammars, although the present version only supports traditional BNF. Our RDP tool[JS97b] supports a restricted version of the permutation operator.

5 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.

A generalised recursive descent (grd) parser can parse any non-left recursive grammar, even in the presence of ambiguity. 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. 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 the 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 that parser using the implementation language's debugger.

As we have discussed above, our grd parsing algorithm has both a prototyping mode, in which ambiguous grammars are fully supported, and a production mode, in which a single derivation is selected. 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[1].

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

We begin by giving a basic definition of formal grammars, to allow us to introduce the terminology and notation that we need. We then give an overview of the operation of a grd parser, a description of the parse trees which are output, and a discussion of the enhancements which improve the parser efficiency. At the end of the section we give a brief discussion of grammars which admit production version grd parsers. A much fuller discussion of the grammar properties required for a production grd parser can be found in [JS97a]. In a forthcoming paper containing a taxonomy of grammars in terms of parsing techniques, we shall discuss the relationship between grd parsers and parsers which use other efficiency measures such as first-match and longest-match.

6 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,

$$A ::= \{aBb, aCC, \epsilon\}$$

is equivalent to

$$A ::= aBb \mid aCC \mid \epsilon.$$

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 of the form $A ::= \tau_A$, 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 $A ::= \tau_A$.

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

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

where $\alpha_{i-1} \Rightarrow \alpha_i$ is a derivation step, $1 \le i \le k$. 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

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

A parser is *conservative* if, when inputting $u \in T^*$ results in success then $u \in L(\Gamma)$. 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.

6.1 Concrete grd parsers

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

GRDP generated parsers contain one parsing function for each rule in the grammar. 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`.

Each alternate production within the rule is mapped onto a `do ... while false` loop statement. This unusual construct (a loop that is guaranteed to execute once) is used in conjunction with the C-language `continue` statement to provide a block of statements from which an arbitrary break may taken.

The loops are then 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` pointer 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 pointer 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.

```
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;
}
```

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

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

6.2 An example grammar

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

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

6.3 GRDP parser trace trees

We use a generalisation of the standard derivation tree to display the results of a parse. Our derivation trees are augmented with nodes that show the start of

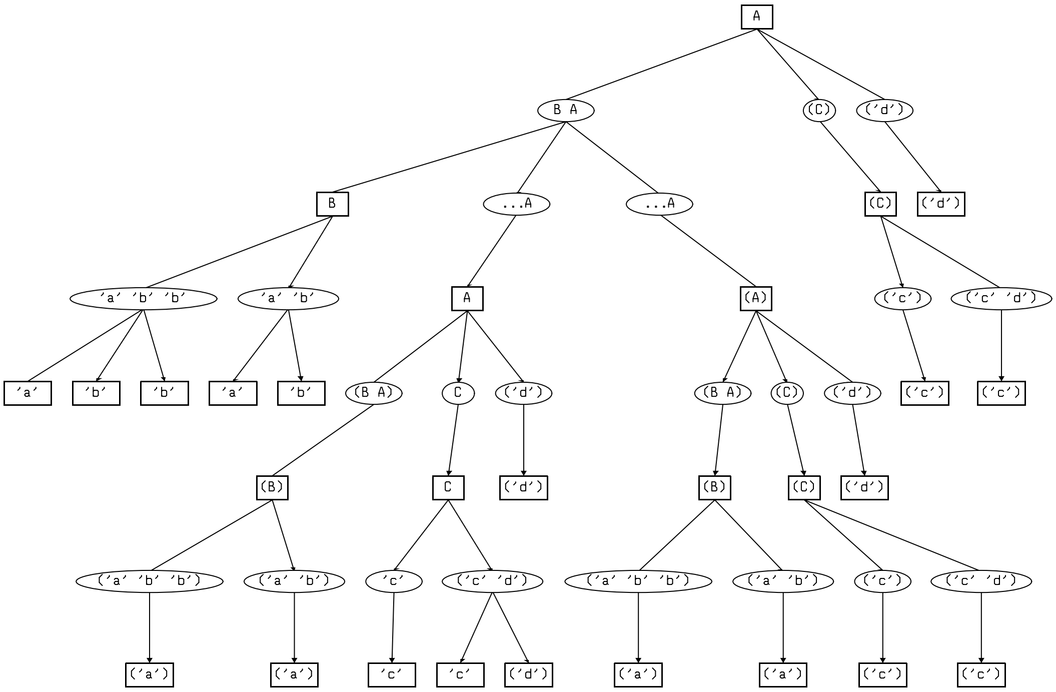

Figure 2 GRDP parser trace tree: example grammar (prototype mode) on string *abbc*

each alternate production and the start of each tail test after a non-terminal has been called. Failed nodes are labeled in parentheses. Figure 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 parse tree.

7 Pruning the search space

We now consider two ways of improving the efficiency of a grd parser. The first is the standard FIRST set checking which 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.

7.1 FIRST set checking

The grd algorithm described so far handles ambiguous grammars by exhaustive searching of the grammar rules. The only concession to efficiency is that the

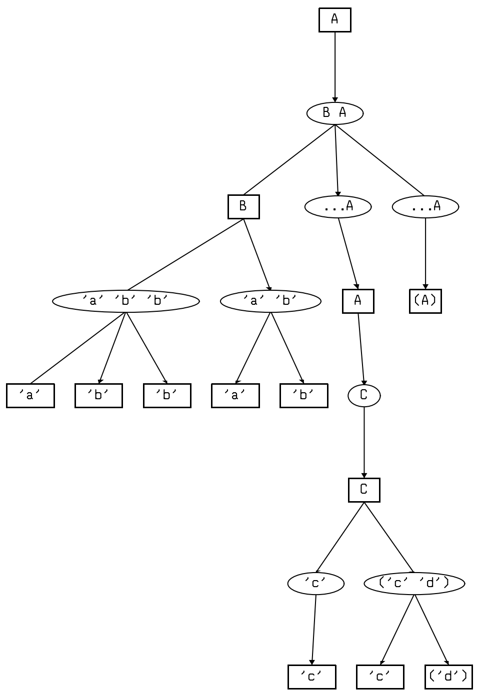

Figure 3 Pruning the search tree with FIRST sets

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 an alternate from its grammar rule 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 GRDP generated parser for this grammar will produce the trace shown in Figure 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:

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

With the FIRST set test, a GRDP generated parser for the above grammar will produce the trace shown in Figure 3 on input *abbc*

The effect of adding such tests is to lop off entire branches within the parser trace tree, which significantly aids efficiency. More particularly, if the user is able to ensure that in each rule the alternates have disjoint FIRST sets (i.e. that the grammar is *left factored*), 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 and calculate 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 grd parsers.

7.2 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 [JS97a]. Here we just give an overview of the basic ideas.

Suppose that a grd parser for the grammar given in the previous section 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 the parser will be able to match both the first and second alternates of the rule.

A parser using the prototyping version 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 version 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 Figure 4 would be produced.

7.3 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 may make it impossible to complete the derivation, while a different choice would have resulted in success. However, it is possible to define a 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 non-terminal A is the set of terminals which can appear immediately after A in a string derived from the start symbol, together

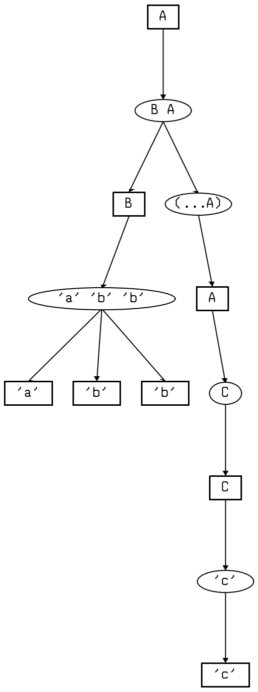

Figure 4 Pruning the search tree with FOLLOW sets

with a special end-of-file symbol \$ if A can appear at the end of a such a string. So

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

where S is the start symbol of the grammar.

Definition A grammar Γ is *follow-determined* if for every $A \in U$, if $A \overset{*}{\Rightarrow} u$ and $A \overset{*}{\Rightarrow} uaw$, 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 then 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.

7.4 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.

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

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 $\{ba^n c \mid n \geq 1\}$. When parsing $ba^n c$, $n \geq 2$, 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. It is also worth noting that the grammar is not LR(1), thus an LR(1) parser generator cannot generate a correct parser from this grammar.

7.5 Theoretical results concerning follow-determinism

The second part of this report [JS97a] describes follow-determined grammars from a theoretical standpoint, and includes results concerning LL(1) and LR(1) grammars as well as a discussion of the impact of left factoring and left recursion removal on follow-determined grammars.

8 GRDP – a generalised recursive descent 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, JS97b, JS97c]. 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 (see [JS97a]) 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 section 5. These trees are output in a form suitable for display with the VCG graph visualisation tool 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`

8.1 Performance of generated parsers

Our emphasis in this report has been on the naturalness of grd as an implementation vehicle for prototyping and refining language specifications, but to be acceptable for production use the generated parsers must be competitive with those produced by other tools. A full investigation of the performance of

GRDP in both prototype and production modes on grammars for ANSI C, ISO Pascal and Oberon is currently being undertaken.

The basic approach in GRDP, of lookahead parser functions, is related to the techniques used in PRECC [BB95]. Initial indications are that in prototyping mode, GRDP has performance similar to that of PRECC on many grammars: the extra overhead of returning complete sets of strings from each parser function being offset by the more efficient iterative implementation within GRDP generated parser functions.

The production mode of GRDP (which returns just one string at each step) is more efficient than the prototype version because in general it performs less searching, and returning a single integer is much faster than constructing a bit-set and converting it to a set array.

The authors of PRECC have published an investigation of the performance of PRECC compared to that of YACC, (both using the FLEX scanner generator) when parsing Oberon code. Oberon is a Pascal family language and we might expect top-down parsers to perform well. In their paper, PRECC is shown to be highly competitive with YACC, although the relative contribution of the parser and scanner to runtimes is not made clear. The authors of PRECC have kindly supplied us with their test suite: we shall repeat their experiment and extend the results to cover GRDP in both modes and also look at ANSI C and ISO-Pascal grammars.

9 Acknowledgements

The authors would like to thank: Terence Parr (the co-author of PCCTS) for helpful correspondence and giving the C++ example used in Section 4.1; Peter Breuer and Jonathan Bower for supplying their Oberon test suite used with PRECC; and Georg Sander for providing of his VCG (Visualisation of Compiler Graphs) tool which was used to draw the figures.

References

[AJ75] A. V. Aho and S. C. Johnson. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8), August 1975.

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

[Bac60] J. W. Backus. The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In R. Oldenburg, editor, *Proc. Internat'l Conf. Information Processing, UNESCO, Paris, 1959*, pages 125–132, London, 1960. Butterworths.

[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.

[Cam93] Robert D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(1–4):85–94, March–December 1993.

[GMS94] Michel Goossens, Frank Mittlebach, and Alexander Samarin. *The LaTeX companion.* Addison-Wesley, 1994.

[HP90] John L Hennessy and David A Patterson. *Computer architecture, a quantitative approach.* Morgan Kaufmann, 1990.

[Inm88] Inmos. *occam2 reference manual.* Prentice Hall, 1988.

[Joh75] S.C. Johnson. Yacc — yet another compiler-compiler. Technical Report Technical Report 32, AT&T Bell laboratories, Murray Hill, N.J., 1975.

[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 2: Some underlying theory. Technical Report TR-97-19, Royal Holloway, University of London, Computer Science Department, October 1997.

[JS97b] Adrian Johnstone and Elizabeth Scott. `rdp` - a recursive descent compiler compiler. user manual for version 1.5. Technical Report TR-97-25, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97c] Adrian Johnstone and Elizabeth Scott. `rdp_supp` – support routines for the `rdp` compiler compiler version 1.5. Technical Report TR-97-26, Royal Holloway, University of London, Computer Science Department, December 1997.

[McC65] Daniel D McCracken. *A guide to FORTRAN IV programming.* John Wiley, 1965.

[MCF+86] J Moussoris, L Crudele, D Freitas, C Hansen, E Hudson, S Przybylski, T Riordan, and C Rowen. A CMOS RISC procssor with integrated system functions. In *Proc. IEEE COMPCON*, San Fransisco, March 1986.

[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.

[Par95] Terence John Parr. *Language Translation Using PCCTS and C++ (A Reference Guide).* Parr Research Corporation, June 1995.

[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.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, 20(11), November 1977.