ARE LR PARSERS TOO POWERFUL?

Philip Machanick
Computer Science Department
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg 2001
South Africa

ABSTRACT The general trend in the development of parser theory is in the direction of exploring implementing methods of increasing power. In particular, ways of improving the efficiency of LR parsers and the generation of LR tables have been receiving a lot of attention. The value of increasingly powerful tools is questioned from the point-of-view of the need to keep definitions of languages understandable to the programmer. Consideration is given to Wirth's contention that recursive descent is the method of choice and alternatives are suggested.

INTRODUCTION

From the point-of-view of exploring theory, LR parsers are the most interesting of the classes in common use. They are the most powerful - LALR and LL are proper subsets of LR - and the challenge is to make it practical to use a full LR parser generator for non-trivial languages.

Some attempts in this direction are briefly surveyed.

The practical implications of the theoretical limitations of LL parsers are considered next. In particular, instances of non-LL constructs are examined. From this discussion, the point that LL parsers do not impose major restrictions on the language designer is made. In fact, it is contended that the restrictions of LL parsers discourage the adoption of language constructs difficult for the human reader to comprehend.

This point is compared with that made by Niklaus Wirth in his Turing Award lecture, concerning the desirability of the recursive descent approach.

In conclusion, consideration is given to how language specifiers ought to be constrained by the tools they have at their disposal.

THE DEVELOPMENT OF LR PARSING

Knuth is credited with the formalization of the LR technique [Knuth 1965], though it is based on earlier techniques, such as operator precedence. In its original form, LR parsing required a very large table compared with other methods (particularly LL). The first breakthrough in simplifying LR parsing was the development of the SLR and LALR methods by DeRemer [Aho and Ullman 1977 p. 243]. Since then, there

has been a considerable amount of research into improved implementation of LALR [Kristensen and Madsen 1981, Park *et al*. 1985] and implementing full LR parsers efficiently [Spector 1981 ; Soisalon-Soininen 1982].

THE LIMITATIONS OF LL PARSERS

LL(*k*) parsers need to be able to determine which alternative production to select according to the next *k* characters in the input. In practice, *k* is usually 1, since any situation which cannot be handled with a lookahead of 1 usually cannot be handled with a larger (fixed-length) lookahead. This is not a theoretical property, but a practical result of the tendency for programming languages to be specified as infinite sets (e.g., identifiers are described as consisting of any number of characters, even though a programmer is unlikely to need one longer than a line on a terminal).

This restriction manifests itself in several classes of constructs which are not LL. The three major classes are illustrated with examples.

The first of these is the dangling else. That such a construct could be a problem was realised by the Algol 60 designers: an **if** was not allowed in the **then** part of another **if** [Naur 1963 §4.5]. To summarise the problem: some convention must be made to decide which **then** an **else** matches in a construct of the form:

1
```
if <boolean expression>
then if <boolean expression>
then <statement>
else <statement>;
```

With a more powerful class of grammar (such as LALR), the ambiguity can be removed by rewriting the grammar [Aho and Ullman 1977 p. 139]. However, the resulting grammar is unwieldy, containing repetitions of the parts of the original simpler grammar. The preferred approach is to use an ambiguous grammar, and some variation on the standard table generation algorithm which allows ambiguity in such cases to be resolved in a natural way. Unfortunately, LL does not lend itself to this approach, while LR does [*ibid*. pp. 225-229].

A related problem is left-factoring. Alternates which can derive a common prefix are not LL, since it is not possible to decide which alternate to choose (unless the lookahead is extended and the common prefix is of a fixed length). In the case of the dangling else, left factoring would not have been any help because it does not remove the ambiguity.

In a simple case, such as:

2
```
<statement>      → <procedure call> | <assignment>
<procedure call> → <identifier> <parameter list>
<assignment>     → <identifier> := <expression>
```

left-factoring is straightforward. Still, the preferred approach is to use semantic routines to choose the appropriate alternate, since a se-

mantic routine is needed at this point in any case for looking up the identifier in the symbol table.

Another non-LL construct (non-LL(k) for any k) is left recursion. Immediate left recursion only specifies repetition, not nesting, and is easy to eliminate [*ibid.* p. 177]:

3 <idlist> → <idlist> , <identifier> | <identifier>

becomes

4 <idlist> → <identifier> <idlist>'
<idlist>' → , <identifier> <idlist>' | <empty>

With the use of { } to enclose an alternate to indicate transitive closure (zero or more repetitions) [Ada 1983 pp. 1-7], repetition can be specified without left recursion. A grammar of the form

5 <idlist> → { <identifier> , } <identifier>

can easily be translated into the grammar 4 [Owen and Reyneke 1982].

Indirect left recursion is more of a problem, however. While an algorithm to eliminate it does exist [Aho and Ullman 1977 p. 179], the problem can no longer be avoided by using transitive closure to write the grammar. Which means readability must be compromised.

DISCUSSION

Are the limitations of LL parsers sufficient to be cause for concern? Why not use LR parsers anyway, since they are more powerful (i.e., LL(k) languages are a proper subset of LR(k) languages for a given k [Nijholt 1982])?

Let us consider the statement made by Niklaus Wirth in his Turing Award Lecture [Wirth 1985 p. 164]:

> ... a tool should be as simple as possible, but no simpler. A tool is counterproductive when a large part of the entire project is taken up with mastering the tool.

What are we really buying by using LR rather than LL? In the first example we considered, the dangling else problem can be solved more easily using an LR parser generator than an LL one. But do we want a language with a construct that is potentially confusing to the reader?

Consider the alternative indentations of the same program fragment:

```
if b1
then
  if b2
  then s
else s
```

```
if b1
 then
   if b2
   then s
   else s
```

What of the second problem we isolated - the need to left-factor a grammar to make it LL? Where the common prefix is obvious, there is no great problem. But if the common prefix is derived after several (non-obvious) steps, the grammar may generate strings which are difficult for the human reader to parse. Even if this is not so, compiler writers using LL parsers and recursive descent (which has similar theoretical limitations) have been able to deal with this sort of problem.

Left recursion is not a major problem if we are prepared to go to the effort of rewriting the grammar. Although this can, in principle, be automated, the drawback of any rewriting preprocessor is that it makes debugging difficult. Be that as it may, using transitive closure instead of recursion to specify repetition has much to recommend it. Not only does it eliminate the problem of immediate left recursion, but it results in a flatter parse tree. Recursion is useful for nested constructs, and for describing precedence in arithmetic expressions. In both these cases, the depth of the parse tree has significance. It describes the structure of the nested construct or shows the order in which parts of the expression are to be evaluated. Where repetition is the only reason for using recursion, adding extra depth to the parse tree is unnecessary. The effect is one of making it more difficult for the reader to spot the parts of the grammar where recursion is really necessary.

Aside from these issues, is it desirable to have left recursion in a grammar? Surely, a person reading a grammar wants to be able to see as easily as possible what terminal strings can be derived. If the grammar is left recursive, it may not be obvious what the first nonterminal is.

Since left recursion can always be eliminated, there are no classes of language construct which cannot be described if left recursion is disallowed.

Taking all these points into account, LL parsers place restrictions on the class of language which can be defined which are desirable from the point-of-view of making the grammar comprehensible. If transitive closure is added as an extension to the BNF notation (not as an extension to its power), the potential for writing readable grammars is increased. Furthermore, the possibility of more efficient parsing involving less recursion is introduced.

What of Wirth's contention that recursive descent is good enough [Wirth 1985]?

The theoretical capabilities of recursive descent are similar to those of LL. The big difference comes in the fact that an LL parser generator transforms the grammar into a table, whereas the recursive descent parser is constructed by hand-translating the productions into a program.

Although the recursive descent parser has the advantage that it can be read by the programmer, and tricks which overcome the limitations of the underlying top-down deterministic model can be hand-coded, it has a significant drawback. The LL parser can be generated directly from the grammar (possibly with some transliteration, which can be achieved with suitable tools). As long as the parser generator and the driver routine interpreting the table are correct, the language parsed will be

that defined by the grammar. Furthermore, the driver routine can be efficiently written (in assembly language, if necessary) once and for all, whereas the speed of the recursive descent parser will be determined by the quality of the implementation of the language it is written in. No matter how well that language is implemented, the recursive procedures making up the parser will incur more overhead than a specialised driver routine accessing a parse table and stack specially designed for this purpose.

RECOMMENDATIONS

Compiler writers obviously remain free to use whatever tools they have at their disposal. However, if language designers follow Wirth's advice and use implementation as part of the design process [*ibid.*], they must not only choose tools which are appropriate to the task in hand, but they must also consider the needs of other implementers.

If LL parsers are sufficiently powerful for most purposes, and constructs which are not LL have the potential to make the grammar unclear, there is a case for designers to avoid using more powerful models.

After all, there are very few LL constructs which are not LALR [Beatty 1982] and none at all which are not LR. So other other implementers would still be free to use their favourite tools.

REFERENCES

Ada 1983.
Ada Programming Language, Military Standard ANSI/MIL-STD-1815A.

Aho, Alfred V. and Ullman, Jeffrey D. 1977.
Principles of Compiler Design, Addison-Wesley, Reading, Massachusetts.

Beatty, John C. 1982.
"On the Relationship Between LL(1) and LR(1) Grammars", *Journal of the ACM*, 29(4) October 1982 (1007-1022).

Knuth, Donald E. 1965.
"On the Translation of Languages from Left to Right", *Information and Control* 8 (607-639).

Kristensen, Bent Bruun and Madsen, Ole Lehrman. 1981.
"Methods for Computing LALR(*k*) Lookahead", *Transactions on Programming Languages and Systems* 3(1) January 1981 (60-82).

Naur, Peter. 1963 (editor).
"Revised Report on the Algorithmic Language Algol 60", *Communications of the ACM* 6(1) January 1963 (1-20).

Nijholt, Anton. 1982.
"On the Relationship Between the LL(1) and LR(1) Grammars", *Information Processing Letters* 15(3) October 1982 (97-101).

Owen, Robert and Reyneke, Chris. 1982.
The Wits Ada Syntax Checker, Honours Project, Computer Science Department, University of the Witwatersrand, Johannesburg, January 1982.

Park, Joseph C., Choe, K. M. and Chang, C. H. 1985.
"A New Analysis of LALR Formalisms", *Transactions on Programming Languages and Systems* 7(1) January 1985 (159-175).
Soisalon-Soininen, Eljas. 1982.
"Inessential Error Entries and Their Use in LR Parser Optimization", *Transactions on Programming Languages and Systems* 4(2) April 1985 (172-195).
Spector, David. 1981.
"Full LR(1) Parser Generation", *SIGPLAN Notices* 16(8) August 1981 (58-66).
Wirth, Niklaus. 1985.
"From Programming Language Design to Computer Construction", Turing Award Lecture, *Communications of the ACM* 28(2) February 1985 (160-164).