LL and LR Translators Need $k > 1$ Lookahead

Terence J. Parr
Parr Research Corporation
San Francisco, CA
parrt@parr-research.com

Russell W. Quong
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907
quong@ecn.purdue.edu

July 6, 1995

Abstract

Language translation is a harder and more important problem than language recognition. In particular, programmers implement translators not recognizers. Yet too often, translation is equated with the simpler task of syntactic parsing. This misconception coupled with computing limitations of past computers has led to the almost exclusive use of $LR(1)$ and $LL(1)$ in parser generators. We claim that use of $k > 1$ lookahead can and should be available in practice, as it simplifies the translator development. We give several examples justifying our arguments.

1 Introduction

Language *translation* is a harder and more important problem than language *recognition*. In particular, programmers implement translators not recognizers. Yet too often, translation is equated with the simpler task of syntactic parsing. This misconception coupled with speed and memory limitations of past computers has led to the almost exclusive use of $LR(1)$ and $LL(1)$ in parser generators.

A *translator* consists of a syntactic parser or *recognizer* augmented with semantic *actions* that are triggered by the recognizer. Semantic actions are typically used for constructing abstract syntax tree nodes, adding entries to the symbol table, or emitting intermediate code. Thus, any program that does parsing, such a compiler front-end, a source code interpreter or a database format converter, is a translator not just a recognizer. Due to the semantic actions, building a translator is harder than building a recognizer, because a translator must execute semantic actions immediately upon seeing the appropriate left contexts.

We claim that use of $k > 1$ lookahead can and should be available in practice, as it simplifies translator development. As this paper is motivated by practice not theory, we justify our arguments via examples; theoretical issues are of secondary concern. We focus on parser generator tools which automatically implement an $LL(k)$ parser [12] [10] or an $LR(k)$ parser [9] [8] given a grammar

specification, because all else being equal, we believe programmers would rather use a parser generator rather than implementing a translator by hand. In addition, programmers should be able to write natural grammars, that reflect the underlying structure of the language and to freely intersperse actions in the grammar.

The amount of lookahead, k, has been limited almost exclusively to $k = 1$ in widely-used parser generators. Two reasons, one practical and one theoretical, explain this situation. However, the first reason is outdated and the second is simply erroneous, as we demonstrate in this paper.

- First, use of $k > 1$ lookahead for a fixed k throughout a parser requires significantly more space and time, in both the parser generator and the resulting parser, than using $k = 1$.
  However, the use of $k > 1$ is feasible now because computers are more powerful and because space-efficient heuristics [11] have been developed that circumvent the intractable nature of parsing with $k > 1$.
- Second, many have felt that $k > 1$ is unnecessary when using an $LR(1)$-based tool because $LR(1)$ equals $LR(k > 1)$ in recognition power [9]. However, this viewpoint is theoretical only and is erroneous in practice.
  - (i) In translation $LR(1)$ is strictly weaker than $LR(k > 1)$. The presence of semantic actions, which must be executed immediately when encountered, makes it impossible to convert an arbitrary $LR(k)$ grammar to an $LR(k - 1)$ grammar, as shown in Section 5.
  - (ii) Even if one were simply recognizing input, converting an $LR(k)$ grammar to an $LR(1)$ grammar is completely impractical, as the resulting $LR(1)$ would be gigantic. More importantly, outwardly the $LR(1)$ grammar would retain almost none of the structure of the original grammar.

Both LR or LL parsers use a finite state machine; their recognition strength is a function of (i) the amount of *context* and (ii) the amount of *lookahead*. We loosely view the context as the information encapsulated in the current parser state. The lookahead is the current set of unparsed tokens available to the parser. The parser uses the lookahead to make parsing decisions. We use k to denote the length of the strings in the lookahead set.

$LR(k)$ recognizers are stronger than $LL(k)$ recognizers because the LR strategy uses more context information. For an LR parser, the context consists of all grammar productions consistent with the previously seen input. This context often includes several "pending" grammar productions. Intuitively, an $LR(k)$ parser attempts to match multiple productions at once and postpones making a decision until sufficient input has been seen. In contrast, the context for an LL parser is restricted to the sequence of previously matched productions and the position within the current grammar production being matched. An $LL(k)$ parser must make decisions about which production to match without having seen any portion of the pending productions—it has access to less context information. Hence, $LL(k)$ parsers rely heavily on lookahead.

As an example, consider using a parser rather than a scanner to recognize integers such as `17` or real numbers such as `17.89`, where each character is a token. An LR parser would have no

problem. It can defer deciding between integer or real until it sees the presence or absence of a decimal point using a natural grammar.

```
lr_gram : digits
        | digits "." digits
        ;
```

In contrast, an LL parser cannot simultaneously parse more than one pending production and cannot see past the common integer prefix to what follows. Thus, the programmer must left-factor the productions into a production matching an integer value followed by an optional fractional value.

```
ll_gram : digits decimal ;
decimal : "." digits
        |
        ;
```

In Section 2, we discuss $LL(k)$'s reliance on lookahead in more detail. While pure $LR(k)$ recognizers have more context information than $LL(k)$ recognizers, we argue in Section 3 that the introduction of semantic actions reduces the amount of available context for an $LR(k)$ parser so that increasing the benefit of lookahead for $LR(k)$ parsers as well. In Section 4, we show that there are languages of practical interest that are more easily specified with $LR/LL(k > 1)$ grammars than by $LR/LL(1)$ grammars. Section 5 briefly discusses the relative strengths of $LR(1)$, $LL(k)$ and $LR(k)$, and Section 6 points out the problem associated with implementing parsers with large lookahead.

2 $LL(k)$ translators

The need for greater lookahead in LL parsers is simply the need for more recognition strength. The inability of LL to automatically left-factor common prefixes of competing rules often forces the programmer to manually left-factor these rules. At best this process leads to unnatural grammars; at worst, it may be insufficient as there are languages that are $LL(k)$ but not $LL(k-1)$ [6]. A larger lookahead k allows the parser to see past more common prefixes, increases recognition strength and minimizes the need for manual left-factoring. Once the programmer has developed an $LL(k)$ grammar, the addition of semantic actions has no effect. Thus, if we can recognize input with $LL(k)$ we can translate it, too. In practice, the use of $LL(k > 1)$ parsers both simplifies grammar development and allows the use of more natural grammars. In Section 4, we provide an example showing $LL(2)$ is much more convenient than $LL(1)$.

3 $LR(k)$ translators

$LR(k)$ is stronger than $LL(k)$ because it provisionally matches multiple productions at the same time—each parser state encodes a set of partially-recognized productions. The parser is not required

to choose between them until the right edge of the winning production. In this section, we show how inserting actions at positions other than at production right edges can affect $LR(k)$'s recognition strength by decreasing its context information. Practical grammars both contain actions "in the middle" of productions, (or more precisely at non-right-edge positions) and require these actions to be executed immediately when encountered. It is these actions which introduce parsing conflicts. Thus for LR parsing, we also urge the use of $k > 1$ lookahead tokens to overcome the lack of context information. We begin by describing how adding actions affects an $LR(k)$ grammar.

While an $LR(k)$ parser can delay parsing decisions, it cannot in general delay action execution. Further, the parser can only execute actions once it has identified which single production to match, because only those actions within the appropriate production should be executed. Actions embedded within a production force the parser to make decisions much sooner than normal; i.e., after the symbols to the left of the action position have been matched. A premature parsing decision is done with less context information and weakens the parser. For example, if an action were placed at the left edge of a production, the $LR(k)$ parser would be forced to decide if that production would succeed without having matched any of it. Unfortunately, this worst case scenario can reduce the strength of $LR(k)$ to $LL(k)$.

The core of our argument for $k > 1$ hinges on the fact that (i) actions must be executed immediately and (ii) actions can occur in the middle or at the left edge of a rule. We now give two examples to empirically prove for our assertion.

(i) Consider translating the following ANSI C type definition and variable declaration.

```
typedef unsigned int boolean;
boolean ignoreCase;
```

To properly translate this code, the symbol table must be updated immediately upon seeing the first `boolean` in the `typedef` declaration so that the subsequent use of `boolean` will be viewed as a type, instead of an unknown identifier. (A standard technique is to have the lexical analyzer return different tokens `TOK_TYPE`, `TOK_VAR`, or `TOK_FN` by examining the symbol table.). The following grammar could be used to recognize the input.

```
typedef_rule : TYPEDEF declaration { add typename} ';' ;
vardef_rule  : declaration { add varname} ;
```

As a subtle point, in an $LR(1)/LL(1)$ parser where the parser always sees one token ahead of what has been matched, the action "*add typename*" must occur *before* the semicolon in the type definition rule in order to update the symbol table before the lexer sees `boolean` in the ensuing statement. If the action execution were delayed until after the recognition of the `';'`, the lexer would not return the correct token type.

(In this paper, we ignore the standard trick of having the lexer use character lookahead to fake $k > 1$ lookahead symbols. This trick only works in restricted cases, such as when the lookahead symbol at $k = 2$ is a single character in length. Worse, this sort of hack results in lexers that are hopelessly interwined with parsers. Anyone who has tried to implement a C++ parser with $LALR(1)$ or $LL(1)$ is familiar with the difficulties.)

(ii) Our next example shows that actions do occur naturally in the middle of rules—some actions cannot be moved to the right edges of a production. Again, our example uses an action to update the symbol table for lexer. In Jim Roskind's [13] precisely-tuned YACC C++ grammar, his comments indicate that parsing scoped names of the form `xxx::yyy` require the lexer to look up the name `yyy` in the namespace for `xxx`. (In C++, the notation `xxx::yyy` refers to name `yyy` in scope `xxx` where the scope is assumed to be global if `xxx` is absent.) Thus the token type for `yyy` is context dependent, as it might be a member variable, member function or type name local depending on the class `xxx`. The following is the rule from Roskind's grammar that handles this situation for global scope overrides.

```
global_scope
    : { direct lexer to look for upcoming name in file scope } "::"
    ;
```

Here, the parser must execute an action to provide context information to the lexer so that it can return the correct token type for the identifier following the scope override operator "`::`". Further, this action must be executed before the parser has had a chance to request the token following the "`::`".

Thus, we have demonstrated that actions must sometimes occur in the middle of a production and that introducing such actions reduces the amount of context information available to an $LR(k)$ parser. This reduction in context is accompanied by a reduction in parsing strength, which can be offset simply with more lookahead information.

4 Example grammars needing $k > 1$

When designing a language, it is a good idea to keep the syntax simple enough to be described easily with an $LR(1)$ or $LL(1)$ grammar; however, it may be the case that the most natural grammar for the language requires $k > 1$. Also, in practice, existing languages have quirks that are best handled via parsers with greater lookahead. (Languages designers are not always familiar with parsing theory).

We now give two examples of standard languages with non-$LR(1)/LL(1)$ constructs. The first shows that increasing k for an $LL(k)$ parser can obviate need to left-factor a grammar. The second example shows that $k > 1$ lookahead symbols are useful for $LR(k)$ grammars even without embedded actions. In the following grammars, tokens are either CAPITALIZED or they are enclosed in single quotes such as ':'.

Example 1: We illustrate the advantage of an $LL(2)$ grammar over an $LL(1)$ grammar for recognizing a fragment of the C language. Consider distinguishing between C labels "`ID :`" and C assignment statements "`ID = ...`", where `ID` represents an identifier token. In the following grammar fragment, rule `stat` requires two lookahead symbols and is easily expressed with an $LL(2)$ grammar. This common language feature is hard to express with an $LL(1)$ grammar because the `ID` token is matched in many different grammar locations. Left-factoring the common `ID` prefix in `stat` and `expr` would results in a much less natural grammar.

```
stat:   ID ":" stat         /* statement label */
    |   expr ";"            /* assignment stat */
    ;
expr:   ID "=" expr
    |   INT
    ;
```

Although manual left-factoring can sometimes solve the problem, it is not ideal for two reasons. First, even if left-factoring yields a reasonable grammar, the $LL(k)$ grammar for $k > 1$ is simply more convenient, which tends to reduce development time. Secondly, while left-factoring might be theoretically possible, it can be practically implausible, as in this example, where `expr` occurs throughout the grammer.

Example 2: Consider specifying a grammar for the "YACC language" which describes the input to YACC itself. Surprisingly, the most natural grammar is $LR(2)$, not $LR(1)$, because the semicolon at the end of a YACC rule is optional, which implies that YACC cannot easily recognize its own input language. Jim Roskind [13], who pointed out this fact to us, also provided the following example YACC input. The problem arises when parsing "$\cdots$ `t t :` $\cdots$".

```
s : A
  | s A
  | t

t : B
  | t B
```

Two symbols of lookahead are required to determine when a rule ends. The difficulty is that we have to scan past a token to check for a following "`:`" to determine whether or not to the current rule is complete. A natural grammar (using extended BNF) for the YACC language would look like the follow

```
yacc_input
    :  ( rule )*
    ;

rule:   RULENAME ':' alt ( '|' alt )* [ ';' ]
    ;

/* This rule requires LL/LR(2) */
alt : ( TOKEN | RULENAME )*
    ;
```

where "`(...)*`" means zero-or-more and "`[...]`" means optional. The "`(...)*`" subrule in `alt` is essentially a loop that consumes `TOKEN`s or `RULENAME`s until something that can follow

an `alt` is seen on the input. The loop will consume tokens until it sees a `'|'` (correct), `';'` (correct) or a `':'` (incorrect). With some work, this natural $LR(2)$ grammar can be transformed into an $LR(1)$ grammar, but the resulting $LR(1)$ grammar would reflect little of the structure of the YACC language. But the use of an unnatural grammar completely negates the advantage of using a high-level grammar. (In fact, if one just wants to recognize the input, a regular expression probably suffices.)

5 Theoretical argument

In this section, we briefly describe how actions may be introduced into an $LR(k)$ grammar at non-right-edge positions and then describe how the effect of action introduction can reduce the strength of $LR(k)$ to that of $LL(k)$ in the worst case.

An $LR(k)$ parser's recognition strength lies with the fact that a given parser state may correspond to multiple grammar positions. Consequently, the parser cannot know which embedded action to execute until it uniquely identifies which surrounding production to reduce; i.e., until it performs a reduce. This implies that productions with actions at non right-edge positions must be "cracked" to force the actions to a right edge. For example, a production of the form

```
a : { action } A;
```

would be transformed into two productions:

```
a : b A ;
b : { action } ; /* production only has action */
```

The introduction of empty productions such as this can introduce parsing conflicts because the parser must decide if that production will succeed before any portion of the cracked production has been matched.

Brosgol [3] showed that a grammar is $LL(k)$ *iff* that grammar augmented with a reference to a unique empty production on each left edge is $LR(k)$. In other words, if a grammar augmented with references to empty productions on each left edge is still $LR(k)$, then the original $LR(k)$ grammar is also $LL(k)$. One may conclude that, while it is unlikely that an action will be required on every left edge, $LR(k)$ can be reduced in strength to that of $LL(k)$ in the worst case.

Carrying the work of Brosgol further, we note that $LR(k)$ must also be stronger than $LR(1)$ in the worst case action placement scenario. This result can be seen by observing that $LR(1)$ is equally strong as $LL(1)$ in the worst case and, since $LL(1) \subset LL(k)$, $LR(1) \subset LL(k)$; by transitivity, $LR(1) \subset LR(k)$. We conclude that $LR(k)$ cannot be arbitrarily rewritten to be $LR(1)$ when actions may be placed arbitrarily in that grammar.

Theory agrees with practice in that converting an $LR(k)$ grammar to be $LR(1)$ (even if the converted grammar were not huge) is not plausible and that $LR(k)$ is significantly weakened when actions are introduced. Increasing the amount of lookahead beyond $k = 1$ for both LL and LR parsers is useful.

A trivial grammar which is $LR(2)$ but not $LR(1)$ due to actions is as follows.

```
start: { printf("X ahead"); } A X
     | A Y
     ;
```

An $LR(1)$ parser that sees the input "A" cannot determine whether or not to execute the action. The grammar cannot be left-factored to reduce the lookahead requirements:

```
start: { printf("X ahead"); } A (X|Y)
     ;
```

The parser in this case would execute the action for both productions. Until the "X" is seen, the action cannot be executed.

6 Implementation Issues

While we have shown $LL(k)$ and $LR(k)$ parser with $k > 1$ to be useful, we have thus far not considered the cost of implementing these parsers. In theory, storing full lookahead information for one decision requires $O(|T|^k)$ space, where $|T|$ is the number of token types. In practice, it is expensive for a parser generator to compute full lookahead for $k > 1$ and the resulting parsers are usually impractically large. Various methods have been examined to obtain $k > 1$ lookahead. For example, [2] and [4] explored using infinite regular languages for lookahead decisions and [1] computed lookahead k-sequences at parse-time rather than statically. Unfortunately, practical and widely-used parser generators have not been developed using these techniques.

We have implemented an effective heuristic [11] called *linear-approximate lookahead* (denoted $LL_1(k)$ and $LR_1(k)$) that retains most of the power of full $k \geq 1$ lookahead, but eliminates the exponential cost of conventional lookahead. Storing linear-approximate decisions requires $O(|T| \times k)$ space, a significant savings over the $O(|T|^k)$ required for full lookahead. By using this heuristic whenever possible, grammars can effectively use reasonably large k ($k = 4$ for large grammars on a 1995 PC/workstation). We construct practical parsing decisions for $k > 1$ in the following manner:

(i) k must be modulated according to the needs of each parsing decision (this technique is well known; [5] suggested this in his paper on $SLR(k)$).

(ii) Most importantly, linear-approximate lookahead must be used to squash the exponential growth for the $k > 1$ case. We rely upon conventional $k > 1$ lookahead decisions only when the approximation is inadequate.

(iii) Even when a full lookahead decision is required, we compute a hybrid approximate/conventional decision in order to reduce the time and space requirements. We take advantage of the fact that, in most cases, approximate lookahead is nearly sufficient—we generate decisions containing an approximate decision plus a few k-tuple comparisons to test for the lookahead sequences that violate the constraints for approximate lookahead.

To illustrate the spirit of linear-approximate lookahead and its attendant time and space savings, consider the following contrived $LL(2)$ grammar.

```
a  :   (A B|C D|F E|H K|J K|A M)
   |   F I
   ;
```

where all symbols beginning with an uppercase letter are token references. The following pseudo-code predicts alternatives of rule "a":

```
if (τ1,τ2) ∈ {(AB),(CD),(FE),(HK),(JK),(AM)} then
     predict first production;
elseif (τ1,τ2) = (FI) then
     predict second production;
endif
```

where τ_i is the i^{th} token of lookahead. Even if a hash table were used to perform the 2-tuple set membership, six 2-tuples would still have to be stored. Alternatively, a series of 2-tuple tests would be both slow and costly in space.

Now consider that, while the first token of lookahead cannot uniquely identify which production to predict, the sets of tokens at lookahead depth $k = 2$ for productions one and two are disjoint. Consequently, the parsing decision can be reduced to the following pseudo-code:

```
if τ2 ∈ {B,D,E,K,K,M} then
     predict first production;
elseif τ2 = I then
     predict second production;
endif
```

The set membership used in this case can be done with a simple bit set test in constant time.

Five years ago, we discovered and implemented linear-approximate lookahead in the widely-used ANTLR [10] parser generator, making $LL(k > 1)$ parsing a practical reality (the first general release for $k > 1$ ANTLR was three years ago). The public-domain parser generator ANTLR, described in [10], is available at

`ftp://ftp.parr-research.com/pub/pccts` .

Finally, we know of a $LR(k)$ parser generator [7] that uses a variant of $LR_1(k)$ that is nearing completion.

7 Conclusion

We have shown a practical need for $k > 1$ lookahead in both LL and LR parser generators. $LL(1)$ is too weak, while the presence of actions reduces the strength of $LR(1)$. Finally, with current computers and heuristic approaches, use of $k > 1$ is feasible.

References

[1] M. Ancona, G. Dodero, V. Gianuzzi, and M. Morgavi. Efficient Construction of $LR(k)$ States and Tables. *ACM TOPLAS*, 13(1):150–178, January 1991.

[2] Manuel E. Bermudez and Karl M. Schimpf. A Practical Arbitrary Lookahead LR Parsing Technique. *Proceedings of the 1986 Symposium on Compiler Construction; SIGPLAN Notices*, 21(7), July 1986.

[3] B.M. Brosgol. *Deterministic Translation Grammars*. Garland Publishing, New York, 1980. Reprint of TR-3-74, Center for Research in Computer Technology, Harvard University, 1974.

[4] Karel Culik and Rina Cohen. LR-Regular Grammars—an Extension of $LR(k)$ Grammars. *Journal of Computer and System Sciences*, 7:66–96, 1973.

[5] Frank DeRemer. Simple $LR(k)$ Grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[6] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin/Cummings Publishing Company, Redwood City CA, 1991.

[7] Josef Grosch (`grosch@cocoab.sub.com`). Private communications, April 1995.

[8] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical Report TR32, Bell Laboratories; Murray Hill, NJ, 1975.

[9] Donald Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.

[10] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-$LL(k)$ Parser Generator. *To appear in Journal of Software Practice and Experience*, 1995.

[11] 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, West Lafayette, Indiana, August 1993.

[12] D. J. Rosenkrantz and R. E. Stearns. Properties of Deterministic Top-Down Grammars. *Information and Control*, 17:225–256, 1970.

[13] James Roskind. Private communications, September 1994.