7-1-1992

A Practical approach to LL(k); LLm(n)

Terence J. Parr
Purdue University, School of Electrical Engineering

Henry G. Dieb
Purdue University, School of Electrical Engineering

A PRACTICAL APPROACH TO *LL (K)*: $LL_M(N)$

TERENCE J. PARR
HENRY G. DIETZ

TR-EE 92-30
JULY 1992

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

A Practical Approach to LL(k): $LL_{\gamma}(n)$[†]

Terence J. Parr and Henry G. Dieb

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
hankd@ecn.purdue.edu

Abstract

In terms of recognition strength, LL techniques are widely held to be inferior to LR parsers. The fact that any LR(k) grammar can be rewritten to be LR(1), whereas LL(k) is stronger than LL(1), appears to give LR techniques the additional benefit of not requiring k-token lookahead and its associated overhead. In this paper, we suggest that $LL(k)$ is actually superior to LR(1) when translation, rather than acceptance, is the goal. Further, a practical method of generating efficient $LL(k)$ parsers is presented. This practical approach is based on the fact that most parsing decisions in a typical $LL(k)$ grammar can be made without comparing k-tuples and often do not even require the full k tokens of lookahead. We denote such "optimized" $LL(k)$ parsers $LL_m(n) \mid m \leq n \leq k$.

† This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013.

grammar, but only at right edges of productions within an *LR* grammar.

Hence, if the user is permitted to insert actions at arbitrary positions within an *LR* grammar, as in YACC [Joh78], rules must be "cracked" to create a reduce corresponding to the placement of the action. For example, given the *LR*(0) grammar (in PCCTS notation):

```
s   :  "x" "x"
    |  "x" "y"
```

inserting two distinct actions, $@_1$ and $@_2$, at the following positions:

```
s   :  "x" @1 "x"
    |  "x" @2 "y"
    ;
```

requires that the LR parser generator restructure the grammar so that the action appears at the right edge of a rule:

```
s   :  s1 "x"
    |  s2 "y"
    ;
s1  :  "x" @1
    ;
s2  :  "x" @2
    ;
```

The unfortunate result of this transformation is that the grammar is no longer unambiguous for *LR*(0) parsing. However, the new grammar is *LR*(1).

Not only is this effect common in constructing *LR*-based translators, but it is also responsible for *LR*(1) NOT being equivalent to *LR*(k). For example, the following grammar would be *LR*(1) without actions, but the actions shown below will cause cracking that results in an *LR*(2) grammar:

```
s   :  s1 "x"
    |  s2 "y"
    ;
s1  :  "x" @1 "x"
    ;
s2  :  "x" @2 "x"
    ;
```

In general, if placing unique actions at every position in an *LR* grammar will result in an unambiguous *LR*(k) grammar, that grammar will also be *LL*(k). The intuitive proof is that the additional strength of *LR* is derived from the ambiguity about the current position in the grammar, by placing an action at every position, we force the current position to be unambiguous at all points in the parse within k tokens of lookahead — the definition of *LL*(k).

In fact, it is sufficient that unique actions be placed at the left edge position in every rule of an *LR* grammar; if the result is *LR* (k), the original grammar with actions must be *LL* (k). A proof of this appears in [PDC92a].

1.1.2. Attributes and Inheritance

As mentioned above, we are interested in translators, not mere recognizers. To effect a transformation, actions must be embedded within the grammar to generate output corresponding to the input phrase recognized. Toward this end, it is useful to have information about each lexeme recognized, and about each nonterminal, available as attributes of the symbols representing them in each grammar rule.

Attributes associated with tokens are implicitly functions of the corresponding lexemes. In contrast, nonterminal attributes are explicitly created and manipulated by actions; they are distinguished from normal attributes by referring to them as inherited attributes.

Nonterminal attributes flow upwards and downwards; rules can compute attributes which are returned upon rule reduction (upward inheritance) or rules can receive information from productions that reference them (downward inheritance). Because *LL* parsers make decisions at the left edge of productions, information can be carried down through each production invocation as well as returned[1]. In contrast, *LR* parsers do not know which set of productions are currently being recognized and are therefore unable to support downward inheritance.

An example of downward inheritance, in PCCTS notation, follows.

```
s   :   t[x] "b" "c"
    |   u
    ;

t   :  "e" << action($0) >>
    ;

u   :   t[y] "d" "e"
    ;
```

Rule t is referenced by both s and u. Until one of those rules has reduced, t does not know which rule referenced it. Conversely, an *LL*(2) parser beginning in rule s would immediately predict the production that would invoke t. Rule s passes some value, x, to rule t via the [x] notation; rule t receives it as $0. Similarly, rule u invokes t with downward inheritance value y. Setting $0 within rule t would set the upward inheritance value for t.

Notice that the $-variable notation is similar to that used in YACC [Joh78], but downward inheritance is not permitted in YACC. so the initial value of $0 is undefined. It is possible for LR-based parsers, such as YACC, to simulate downward inheritance by first building a syntax tree of the entire input and then traversing the tree in a manner similar to an *LL* parse [PuC89].

[1] A good overview of attribute handling in *LL* (*k*) parsers is given by Milton and Fischer in [MiF79].

However, the efficiency of building a large tree and traversing it seems highly questionable. Further, downward-inherited attributes can be used in semantic actions that change the syntax accepted by an *LL* parser, whereas the *LR* tree-building scheme cannot.

Downward inheritance provides significant power because a rule can decide which rule referenced it. For example, in a rule which recognizes variable declarations for a programming language, actions can be a function of an inherited attribute. This attribute could carry information as to whether local or global variables were about to appear on the input stream. This implies that one rule could match the same syntactic structure while dealing with many different semantic situations. In general, foreknowledge about what production is being recognized proves useful in practice with regards to triggering actions. It can also be used to implement non-strong *LL(k)* parsers by passing in the "local" $FOLLOW_k$ set for the rule being referenced; i.e. the $FIRST_k$ of what immediately follows the rule being referenced.

1.13. Grammar-oriented Symbolic Debugging

When grammars become large or when they are augmented with actions, it is often necessary to trace the execution of the parser. The one-to-one mapping of parser state to grammar position using *LL* parsing makes it easy to trace the recognition process. For example, a user could specify that the parser should consume input until an arbitrary position in the grammar has been reached. In many systems, this can even be accomplished without recompiling the parser — simply using a standard symbolic debugger.

In contrast, to set a "breakpoint" at a grammar position within an *LR* parse, it may be necessary to "crack" the grammar as described in section 1.1.1. This would require that the parser be recompiled.

1.1.4. Practical Efficiency Issues

LL parsing uses its stack the same way recursive subroutine calls use their stack. Many computers have been designed to make the recursive-call stack use efficient. Hence, by generating the parser as a recursive "program" rather than a simulated automaton, some speed advantage is gained. This also aids in the handling of attributes, which act like function arguments, local variables, and return values. Some additional advantages deriving from program, versus automaton, implementation are given in sections 3.3 and 4.4.

Although [Rob90] obtains some of these advantages for *LR* parsers by using "recursive ascent," the mapping of *LR* stack use onto function call/return is not obvious and is rarely used.

1.2. Why LL(k) Is Not Used

Because *LL* parsers make decisions on the left edge, lookahead size directly effects parser recognition strength. A parser with k tokens of lookahead can choose between all productions (which are recognizable at a particular point) that have common prefixes of length $k-1$ or less. The degenerate case occurs when each alternative at a decision point begins with a different token; only one token of lookahead is required to distinguish between them (hence *LL(1)*) since

there are no common prefixes. Since *LL*(0) can never distinguish between alternatives, *LL*(0) works only when there are no choices to be made.

LL's left-edge decision rule and its dependence upon lookahead introduce a number of difficulties. This section addresses parser run-time issues related to lookahead and discusses translator development. Fortunately, in practice the problems mentioned here are surmountable with experience and rarely present more than an inconvenience to the developer.

13.1. Expressive Power/Ease of Grammar Construction

Consuucting a recognizer for a given language is generally easier with an *LR* parser generator such as YACC [Joh78] than it is for an *LL* parser generator because there are fewer restrictions.

For example, *LL* grammars may not contain left-recursion and alternatives may not specify common token prefixes of length $\geq k$ where k is the parser lookahead size. Although it is possible to transform grammars to remove left-recursion [AhU79], prefixes can't always be left-factored to remove *LL* ambiguities. Thus, some *LR* grammars have no *LL* equivalent.

The catch is that, as per section 1.1.1, inserting actions in your non-*LL* *LR* grammar can cause "cracking" so that the resulting grammar is neither *LL* nor *LR*. The fact that it is easy to accidentally "break" a grammar in this way is the prime source of frustration for many users of *LR* parser generators.

Difficulties with *LR* action insertion have led to the idea that all languages should be designed to be parsable with both *LL* and LR techniques. Since most languages are designed this way, the additional strength of *LR* parsing isn't often utilized.

13.2. Comparisons to k-Tuples

Because LL is weaker than *LR*, it may be necessary to use larger values of k for *LL* than for *LR*.

To chose an alternative production, an *LL*(k) parser must compare the next k tokens with all possible k-token tuples that could be in the $FIRST_k$ set for each alternative. The obvious problem is that, as k becomes large, the number k-tuples to compare with can be as large as v^k where v is number of tokens in the grammar vocabulary ($v = |V|$). Further, each k-tuple comparison may require k single-token comparisons.

This is a serious problem in traditional *LL*(*k*) parsing, but the proposed $LL_m(n)$ avoids using k-tuples wherever possible.

133. k-Token Lookahead with Symbol Table Interactions

The dependence upon lookahead in *LL* parsers presents more than a run-time complexity problem. Translators generally need to deal with lexemes that can represent different tokens depending on their context in the input. For example, an alphanumeric string may be a label name in one scope, a variable in another, and a user-defined type in yet another context; typically,

the token for such a lexeme is determined by lookup in the symbol table. Many language grammars are highly ambiguous without this kind of context-sensitive lexeme-to-token mapping.

Unfortunately. this technique presents a problem when a token is looked up in the symbol table before the correct context has been entered by the translator — a common occurrence when k tokens of lookahead are used. Consider the following grammar fragment, which recognizes a sequence of simple type and variable definitions.

```
s   :  t ";" s
    |
    ;

t   :  USER_TYPE WORD          << define user-type variable >>
    |  "int" WORD              << define integer variable >>
    |  "typedef" "int" WORD    << add new type to symbol table >>
    ;
```

In the above, uppercase words and quoted elements are tokens. Although the grammar is actually $LL(1)$, a traditional $LL(2)$ parser would fail to correctly recognize input.

The problem arises when a user type is defined and it is followed immediately by variable definition using that new type, the lexical analyzer may not find that new type in the symbol table.

```
typedef int boolean;
boolean bvar;
```

If the parser always maintained two tokens of lookahead, it would obtain lookahead of `; WORD` rather than `; USER_TYPE` — calling the lexical analyzer for a token for `boolean` before the `typedef` had changed the symbol table.

This type of problem can be avoided in many cases by restricting symbol table lookup from the lexical analyzer or by delaying lookahead fetches until tokens are needed by a parsing decision. The $LL_m(n)$ approach effectively delays the fetches.

2. Practical $LL(k)$ Parsers

$LL(k)$ parsers make decisions whose worst case complexity is exponential in k as discussed in section 1.2.2. Fortunately, one can parse $LL(k)$ grammars significantly faster than the worst case would imply without resorting to large automatons.* In the best case, one can decide between two alternative productions in time proportional to k (even constant time if early error detection is not a concern).

The key to constructing practical $LL(k)$ recognizers is the observation that it is impossible to construct a grammar for which every parsing decision requires k tokens of lookahead for k>1. In fact, the vast majority of parsing decisions for a typical grammar can be made either

[2] Note that, overall, parsing is considered $O(n)$ for inputs of length n for $LL(k)$ parsers [AhU72].

with no lookahead or with one token of lookahead. An additional simplification often can be made because, in the few cases in which it is necessary to look $n \leq k$ tokens into the future, it is rarely necessary to compare the next n tokens of input against all possible n-tuples for that production. By constructing a parser that can dynamically switch between different lookahead depths and comparison structures, these grammar properties can be exploited to reduce both parser size and parsing run-time.

A parser generator must analyze each grammar decision point and synthesize a parser decision rule which uses as few tokens of lookahead as possible and performs a reasonably small number of comparisons. We will introduce the notion of $LL_m(n)$ which describes the class of languages recognizable by different parser decision templates. No claim or proof of optimality regarding our decision templates is offered in this paper, but we have implemented an efficient $LL(k)$ translator writing system called PCCTS (Purdue Compiler Construction Tool Set) [PDC92] which uses the reduction techniques discussed in this section. For the moment, we will ignore the method by which an LL(k) grammar decision point may be analyzed to obtain k-tuples representing the possible production token prefixes in an effort to concentrate on parser construction.

Also, for simplicity, we will consider only BNF grammars since Extended BNF grammars are easily translated to BNF [AhU79]. To avoid some of the lookahead problems discussed in section 1.2.3, we assume that parsers for given BNF grammars delay lookahead fetches as long as possible.

Our discussion of LL grammars will be limited to those which satisfy the Strong LL condition since all LL grammars have Strong LL equivalents. Strong grammars have the property that the "global" FOLLOW, versus a "local" FOLLOW, may be used to predict alternatives within a rule; this property typically leads to smaller parsers. The "global" $FOLLOW$ set for a rule is the set of all tokens that can possibly follow any reference to that rule. The "local" $FOLLOW$ is the set of tokens that can follow a specific reference to that rule.

The decisions made by our LL(k) parsers will be binary in nature; a sequence of $a-1$ decisions is needed to uniquely determine which of a alternatives applies.

This simplification is one commonly made in computer hardware to avoid N-ary decisions, but is done here to simply our presentation. To reduce LL(k) decision complexity, decisions can easily be made in binary tree search fashion which drops the number of decisions from $a-1$ to $\log_2 a$. It is also possible to reduce this decision to application of a hash function to the next k tokens yielding a complexity of $O(1)$ to choose a production; however, the hash table would often be huge and determining a good hash function would be difficult. In addition, it would be very difficult to correctly interpret hashed entries for k token lookaheads that involve symbol table interactions like those discussed above in section 1.2.3. The $a-1$ decisions discussed here can themselves be optimized heavily, but each take $f \times k$ token comparisons in the worst case where f is is the number of k-tuples in a particular $FIRST_k$ set (yielding a worst case of $O(|V|^k)$ where $|V|$ is the number of terminals).

2.1. Background

Before examining parser construction in detail, a few definitions and bit of language theory are in order. Our notation is based upon past works in language theory [SiS82] [FiL88] [AhU79], but we will present the material in more practical, less rigorous manner.

As in the above examples, words in uppercase and quoted regular expression represent terminals; lowercase words represent nonterminals. Actions are enclosed in European quotes (<<, >>) and rules are defined in a fashion similar to YACC [Joh78] to bring a sense of familiarity:

```
s   : alternative1
    | alternative2
      ...
    | alternativea
```

where alternative$_i$ is a sequence of terminals and nonterminals.

We shall denote the set of terminals in a grammar as V for vocabulary where v*, V^+ and V^k represent sequences of length "zero or more," "one or more" and k respectively. N is the set of nonterminals defined by the productions and has the same closures to V (e.g. N^+). The language generated by a grammar, G, is $L(G)$ — the set of all terminal sequences (strings) that can be recognized by G beginning at the start symbol, s, in zero or more derivation steps; formally, the set $\{\omega \in V^* \mid s \Rightarrow^* \omega\}$. We shall consider the size of a grammar, |G|, to be proportional to the number of positions in G; i.e. roughly the number of references to tokens or nonterminals. G is LL(k), or $LL(k)$-decidable, if an LL(k) parser can be constructed to deterministically recognize $L(G)$; in other words, a parser that correctly predicts which productions to apply from the left-edge using a maximum of k tokens of lookahead.

The concepts of $FIRST_k$ and $FOLLOW_k$ are fundamental to determining $LL(k)$-decidability and constructing parsers. $FIRST_k(\omega)$ is simply the set of strings of length k that can possibly be recognized by ω where $\omega \in V^* \cap N^*$ [AhU72]. $FIRST_k$ sets are typically defined as a set of k-tuples; we introduce FIRST trees as a practical alternative in section 3.1. If we know the $FIRST_k$ set for each production in an alternative list, one can determine which production to apply given k tokens of lookahead (unless the FIRST sets are not disjoint). For example, $FIRST_3$(A B C d) is the 3-tuple (A, B, C). Ifsomerule dwere:

```
d   : D | E ;
```

$FIRST_4$((A B C d)) would be the set {(A, B, C, D), (A, B, C, E)}. Whichimplies that $FIRST_1$(d) is the set {D, E}. If $FIRST_k$ is required for some rule that can only supply token strings of length n where $n \leq k$, $FOLLOW_{k-}$ is required to complete the computation of the $FIRST_k$ set.[3] In the case of $k=1$, any rule that is nullable (has an empty production) requires the

[3] Note that our $FIRST_k$ sets differ slightly from the norm in that our $FIRST$'s include the $FOLLOW$ set when necessary so that $FIRST_k$ truly represents the set of k-tuples that can begin a production.

$FOLLOW_1$ set for that rule. $FOLLOW_k(t)$ for some rule `t` is the sequence of tokens that can possibly be matched after some reference to rule `t`. For instance, consider the following grammar:

```
s   : A t B C
    | t B D

u   : t C D
    ;

t   : T ;
```

$FOLLOW_2(t)$ is the set of 2-tuples {(B, C), (B, D), (C, D)}. $FOLLOW_k(t)$ can be defined in terms of FIRST sets; it is $FIRST_k(\omega)$ for all $\omega \in V^* \cap N^*$ that immediately follow references to `t`. The algorithm for computing FIRST and FOLLOW sets outlined in section 3 takes advantage of this definition to simplify set construction.

A grammar decision is considered $LL(k)$-decidable if, for all productions in the alternative list, the corresponding $FIRST_k$ sets are disjoint. If all decisions are $LL(k)$-decidable, the grammar is $LL(k)$-decidable (i.e. $LL(k)$). Decisions that are not $LL(k)$-decidable are considered ambiguous.

The definitions presented in this section are central to the discussions given below since parsing decisions are generically of the form

```
rule()
{
    if ( (τ1, τ2, ..., τk) ∈ FIRSTk(alternative1) ) {
        recognize alternative1
    }
    else if ( (τ1, τ2, ..., τk) ∈ FIRSTk(alternative2) ) {
        recognize alternative2
    }
    ...
    else if ( (τ1, τ2, ..., τk) ∈ FIRSTk(alternative,) ) {
        recognize alternative,
    1
}
```

for each rule present in the grammar. The following sections describe multiple parsing templates and characterize when they can be used.

22. $LL(n) \mid n \in [0..k]$

Most parsing decisions require at most one token of lookahead, but there are grammar constructs for which n tokens are needed where $n \leq k$. However, there is no need to degrade parsing speed by forcing all decisions to use the amount of lookahead required by the most complex construct in the grammar. This section characterizes those situations; i.e. those situations that need $n \leq k$ tokens of lookahead and compare n tokens of lookahead against n-tuples to determine which

alternative applies.

The following grammar contains both *LL*(1) and *LL*(2) constructs:

```
s   :  "a" "b"
    |  t
    ;

t   :  "c" "b"
    |  "a" "d"
    ;
```

The first and second alternatives of rule `s` require two tokens of lookahead to determine which alternative applies whereas alternatives one and two of rule `t` can be distinguished with only one. An efficient parser would use only as much lookahead as necessary. For example, the above grammar could be loosely translated to pseudo-C in the following way (ignoring error conditions).

```
s()
{
    if ( (τ1, τ2) == ("a", "b") ) {
        match("a");
        match("b");
    }
    else if ( (τ1, τ2) ∈ {("c", "b"), ("a", "d")} ) {
        t();
    }
}


t()
{
    if ( τ1 == "c" ) {
        match("c");
        match("b");
    }
    if ( τ1 == "a" ) {
        match("a");
        match("d");
    }
}
```

where (τ_1, τ_2) is a tuple containing the next two tokens of lookahead, τ_i is the i^{th} lookahead token and $(\tau_1, \tau_2) == (\text{"a"}, \text{"b"})$ represents a tuple comparison.[4] Decisions within the same list of alternatives can even be made using different amounts of lookahead. For example, if we extended rule `s` to include another alternative,

[4] Here, we use the notation Ti == *"string"* to represent comparison of the token values, rather than the pointer comparison suggested by the usual C interpretation of the construct.

```
s   :  "a" "b"
    |  t
    |  "q"
    ;
```

we would still handle alternatives one and two as before, but alternative three could be predicted using only one token of lookahead.

```
s()
{
    if ( (τ1, τ2) == ("a", "b") ) {
        match("a");
        match("b");
    }
    else if ( (τ1, τ2) ∈ {("c", "b"), ("a", "d")} ) {
        t();
    }
    else if ( τ1 == "q" ) {
        match("q");
    }
}
```

thus saving a token comparison

When many tokens of lookahead are required to predict a production, the tuples in FIRST, may have many prefixes in common. If n tokens are needed to disambiguate a decision, there must be at least one token sequence of $n-1$ that is common to two or more productions in the altemative list; which leads one to believe that the FIRST, set of an individual production may also have n-tuples with common prefixes. Just as we left-factored grammars in section 1.2.1 to remove ambiguities, we can left-factor parsing decisions to remove redundant comparisons as a practical matter. To illustrate the usefulness of this technique, consider:

```
s   :   t
    |  "a" "b" "e"


t   :  "a" "b" "c"
    |  "a" "b" "d"
    ;
```

The parser for rule s would normally compare (τ_1, τ_2, τ_3) against the two 3-tuples of $FIRST_3(t)$:

```
("a", "b", "c")
("a", "b", "d")
```

which would compare τ_1 and τ_2 against "a" and "b" (respectively) more than necessary. If those two n-tuples were left-factored, a more efficient parsing rule could be obtained:

```
s()
{
    if ( τ1 == "a" && τ2 == "b" && τ3 ∈ {"c", "d"} ) {
        t();
    }
    else if ( ... ) {
        match("a");
        ...
    }
}
```

The savings becomes even more evident when larger grammars are considered.

This section described how varying degrees of lookahead can be used to generate smaller and more efficient parsers. More impressive reductions can be achieved by comparing m-tuples rather than n-tuples where (m $<n$).

23. $LL_m(n)$ | m *and* $n \in [0..k]$ *and* $m \leq n$

Not all parsing decisions have to be identical in nature and general enough to handle any construct in the grammar. As demonstrated above, one can reduce decision complexity by varying the amount of lookahead for each decision even if the same parsing decision template is used. This section describes the situations for which multiple parsing templates can be used in conjunction with our strategy of using minimal amounts of lookahead. We introduce the notion of $LL_m(n)$ as a means of describing the different parsing templates.

$LL(n \leq k)$ parsers must consider a number of n-tuples for each decision. The concept of left-factoring presented above reduces decision complexity by observing that the comparison of n-tuples with a common prefix of length m may be broken down into one m-tuple comparison followed by a number of "$n-m$"-tuple comparisons. The example given in 2.2:

```
s   :   t
    |  "a" "b" "e"
    ;

t   :  "a" "b" "c"
    |  "a" "b" "d"
    ;
```

was parsed using a decision that was left-factored. i.e.

```
s()
{
    if ( τ1 == "a" && τ2 == "b" && 5 ∈ {"c", "d"} ) {
        t0;
    }
    ...
}
```

It can be reformulated as a 2-tuple comparison followed by two 1-tuple comparisons while still using three tokens of lookahead.

```
s()
{
    if ( (τ1, τ2) == ("a" "b") && τ3 ∈ {"c", "d"} ) {
        t();
    }
    ...
}
```

The first, left-factored, decision used tuples of size one and used three tokens of lookahead; hence, it is considered $LL_1(3)$. The reformulation uses tuples of size at most two and needs three tokens of lookahead resulting in an $LL_2(3)$ parsing decision. Left-factoring is an implementation detail in actuality but is also a special case of $LL_m(n)$; e.g. when all n-tuples have a common prefix of $n-1$ tokens, left-factoring is really a $LL_1(n)$ decision. $LL_m(n)$ is much stronger than simple left-factoring because it handles situations where n-tuples have no common prefixes.

Formally, $LL_m(n)$ with $m,n \in [0..k]$ and $m \leq n$ is contained in $LL(k)$. An $LL_m(n)$ parsing decision examines permutations of at most m lookahead tokens and looks no further than n tokens into the "future." $LL_k(k)$ examines k-tuples of at most k tokens in the future and therefore represents familiar $LL(k)$.

Creating an efficient LL(k) parser amounts to determining the minimum m and n needed to construct each parsing decision. We constrain m to zero, one or k here because $1 < m < k$ is rarely needed and can be handled by $LL_k(k)$ thus simplifying our discussion without sacrificing generality.[5] This constraint arises naturally from the fact that we can perform n set memberships much faster than we can compare multiple n-tuples. Section 4.4 describes how a set membership operation can be performed in constant time for a fixed vocabulary, V.

In the following sections, we will show that efficient parsers can be constructed for $LL(k)$ grammars using a combination of $LL_0(0)$, $LL_1(1)$, $LL_1(k)$ and $LL_k(k)$ where k is some user-defined maximum.

23.1. $LL_0(0)$ parsing decisions

Tokens occumng consecutively in a production can be recognized without a parsing decision because the expected stream of input tokens can be statically determined; hence, token sequences within a single production are $LL_0(0)$. For instance,

```
s   :  "a" "b" "c"
    ;
```

defines a rule called s which matches three tokens in the sequence a b c. A parser generated using the C programming language would resemble the following code fragment.

[5] To create optimal parsing decisions, $LL_m(n)$ for $1 < m < k$ would have to be considered

```
s()
{
    match("a");    /* match is a macro that checks for invalid tokens */
    match("b");
    match("c");
}
```

No parsing decisions are required and *code* execution simply flows through the three **error detection macros.**

23.2. $LL_1(1)$ parsing decisions

This class of decisions is the most common and is equivalent to $LL(1)$. Any decision that can be **made** by examining only one token of **lookahead** falls into this category. Decisions are always made in constant time since they represent set membership operations in the worst case. For example,

```
s   :   t
    |  "a" "b"
    ;

t   :  "x" "y"
    |  "z"
```

Rules s and t are $LL_1(1)$. Rule s could be parsed via:

```
s()
{
    if ( τ1 ∈ {"x", "z"} ) {
        t();
    }
    else if ( τ1 == "a" ) {
        match("a");
        match("b");
    }
}
```

233. $LL_1(n)$ parsing decisions

The class of decisions represented by $LL_1(n)$ is the most important because, when applicable, it reduces decision complexity from $O(|V|^k \times k \times a)$ (for a alternatives) to $O(n \times k \times a)$ where $n \leq k$. It is primarily because of this decision template that $LL(k)$ parsing becomes practical.

Consider a **production/rule** with f n-tuples in its ***FIRST,*** set. Let Λ_i be the set of tokens collected **from** the i^{th} position in each of the f n-tuples. Also, let Λ_i^j represent the set of tokens collected **from** position i **from** the ***FIRST,*** tuples for ***the*** j^{th} production. Under certain circumstances. a parsing decision using Λ_i sets can be used to predict productions; **i.e.**

```
if ( τ1 ∈ Λ1^1 && ... && τn ∈ Λn^1 ) {
     ...
}
else if ( τ1 ∈ Λ1^2 && ... && τn ∈ Λn^2 ) {
     ...
}
...
if ( τ1 ∈ Λ1^a && ... && τn ∈ Λn^a ) {
     ...
}
```

where a is the number of alternatives. Each if expression requires n set membership operations and therefore has complexity which is linear in the size of the lookahead required to make the decision. The situation in which this type of decision can be applied is characterized by

$$\bigcap_{j=1}^{j=a} \Lambda_n^j = \varnothing \qquad \text{(C1a)}$$

and

$$\bigcap_{j=1}^{j=a} \Lambda_i^j \neq \varnothing \qquad i = 1..n-1 \qquad \text{(C1b)}$$

for some n. Which implies that an n exists for which τ_n can be used to distinguish between all a alternatives. Condition (C1b) indicates that each production has at least one sequence with a token appearing at τ_i that is common to all productions in that alternative; τ_i cannot be used to predict which production applies. (C1b) guarantees that n is the minimum lookahead needed for this template.

To find the n in conditions (C1a) and (C1b), one simply considers larger and larger amounts of input (beginning at $n=1$) until a satisfactory n is found. As an example, consider the following grammar

```
s   :  "a"  t  "d"
    |  u  "b" "f"
    ;

t   :  "b" | "c" ;

u   :  "a" | "e" ;
```

which is $LL(3)$. Rule s yields a $FIRST_3$ set of

```
{("a", "b", "d"), ("a", "c", "d")}
```

for alternative one and

```
{("a", "b", "d"), ("e", "b", "d")}
```

for alternative two. The Λ_i sets can easily be computed:

Λ^1 = {{"a"}, {"b", "c"}, {"d"}}
Λ^2 = {{"a", "e"}, {"b"}, {"f"}}

The first two sets have tokens in common, but Λ^1_3 and Λ^2_3 are disjoint. The following function would parse rule s.

```
s()
{
    if ( τ1 == "a" && τ2 ∈ {"b","c"} && τ3 == "d" ) {
        match("a");
        t();
        match("d");
    }
    else if ( τ1 ∈ {"a","e"} && τ2 == "b" && τ3 == "f" ) {
        ...
    }
}
```

where membership operations for singleton sets have been converted to single token comparisons. This parsing template is $LL_1(3)$ because tuples of size at most one were considered using at most three tokens of lookahead.

Note that if the input is guaranteed to be a valid sentence in $L(G)$, τ_n is sufficient to parse the input correctly when (Cla) holds; e.g.

```
s()

    if ( 23 == "d" ) {
        ...
    }
    else if ( τ3 == "f" ) {
        ...
    }
}
```

$\tau_1 \ldots \tau_{n-1}$ can also be ignored when the input is invalid if the user does not care how soon an error is detected. If no $n \leq k$ exists which satisfies (Cla) and (Clb), a more powerful decision template is needed (or. the grammar is not LL(k)).

23.4. $LL_n(n)$ parsing decisions

If an LL_1(n) parsing template cannot be used, either the language cannot be specified unambiguously (e.g. dangle-else clause), the grammar is ambiguous, k is not large enough, or an n-tuple, which was artificially introduced because of the A sets, can be recognized by more than one decision. If the grammar is ambiguous, no $LL(k)$ decision for any k will resolve the situation.

Increasing k indefinitely in search of a τ_k that disambiguates a decision using $LL_1(k)$ is infeasible; typically, $k \leq 5$ is sufficient lookahead that if a satisfactory τ_k is not found, $LL_n(n)$ should be considered.

$LL_1(n)$ reduces f n-tuple comparisons to n token comparisons **and/or** set memberships where f is |FIRST$_n$| for a production. However, it does so at a cost. Using A sets to represent n-tuples is efficient, but introduces artificial n-tuples that were not actually present before "compaction." This poses no problem unless these artificial tuples conflict with a valid or artificial tuple from a FIRST$_n$ set from another production; in which case, $LL_1(n)$ cannot be used to predict productions in that alternative list. For example,

```
s   : t
    | "a" "d"
    ;

t   : "a" "b"
    | "c" "b"
    | "c" "d"
```

Λ^1 for rule `s` is `{{"a","c"}, {"b","d"}}` and Λ^2 is `{{"a"}, {"d"}}`; hence, $\Lambda^1 \cap \Lambda^2$ = `{{"a"}, {"d"}}` which violates conditions (Cla) and (Clb) **from** section 2.3.3 when n =2. An $LL_1(2)$ decision for alternative one in `s` would recognize the four 2-tuples:

```
("a", "b")
("a", "d")
("c", "b")
("c", "d")
```

i.e. it would test:

```
if ( τ1 ∈ {"a","c"} && τ2 ∈ {"b","d"} ) {
    ...
}
```

The sequence `"a" "d"` could therefore be recognized by both **alternatives** in `s`. Although τ_1 can be `"a"` and τ_2 can be `"d"` for alternative one, `("a", "d")` is **not** a valid sequence for that alternative (**i.e.** $\notin FIRST_2(t)$).

To resolve sequencing problems like this, n-tuples must be considered. If further grammar analysis indicates that the conflicting tuples are indeed valid, the parsing decision is $LL(n)$-undecidable (ambiguous). If the conflicting tuples are purely artificial and no valid tuples overlap across $FIRST_n$ sets, then the parsing decision is $LL(n)$-**decidable** and an $LL_n(n)$ decision can be used to **correctly** predict productions in that alternative list.

One is not left with the prospect of testing f n-tuples, however. Combining $LL_1(n)$ with a few tuple comparisons can be more efficient than the straightforward approach; although it is not always the case. $LL_n(n)$ decisions can be made by augmenting $LL_1(n)$ decisions with a test that

prevents the artificially generated n-tuples from being recognized. For instance, rule s above can be passed in the following manner:

```
s()
{
    if ( τ1 ∈ {"a","c"} && τ2 ∈ {"b","d"} && !(τ1 == "a" && τ2 == "d") ) {
        ...
    }
    else if ( !(τ1 == "a" && τ2 == "d") ) {
        ...
    }
}
```

In this case, it is more efficient to combine $LL_1(n)$ and one 2-tuple comparison (4 simple compares) than to perform 3, 2-tuple comparisons (6 simple compares).

Because $LL_1(n)$ analysis is efficient and handles the majority of cases, it should be performed before $LL_n(n)$. Even if $LL_1(n)$ is insufficient to form a valid parsing decision, the results of its analysis are still of value. The cost of constructing $LL_n(n)$ sets can be reduced by constraining the traversal of the syntax diagram to those paths whose edge labels (tokens) are in the set of possible ambiguous sequences. Two alternative productions with A sets, Λ^i and Λ^j, are ambiguous upon at most those sequences described by the set $h^i \cap \Lambda^j$. Only sequences in this set need be considered since they represent the set of sequences that invalidates our $LL_1(n)$ parsing decision. The $h^i \cap \Lambda^j$ set is pruned by our $LL_n(n)$ analysis to remove all artificial token sequences yielding a set containing only those sequences that are $\in L(G)$.

When n-tuple comparisons are required for an $LL_n(n)$ decision, left-factoring can be used once again reduce then number of token comparisons. However. a bigger savings is derived from the use of trees to represent tuples as is discussed in 3.1.

This section introduced the notion of $LL_m(n)$ as a method of reducing $LL(k)$ parsing complexity. Determining the optimal parsing expression can be accomplished by exhaustively testing various values of m and n; but, it proves unnecessary in practice because most decisions are $LL(1)$. Any good solution for an LL (k) decision is sufficient since they occur so infrequently.

This paper focuses upon LL (k) parser construction but one cannot ignore the issue of LL (k) grammar analysis since one cannot determine $LL(k)$-decidability or generate parsers without it. We provide an overview of our analysis method in the next section so that the reader may gain some insight into the problems associated with computing *FIRST* sets.

3. Analysis

To build a parser from a given grammar, a parser generator must compute *FIRST* sets in order to construct parsing decisions and to determine LL-decidability. The way in which grammars and *FIRST* sets are represented, have an enormous impact on algorithm simplicity and parser construction. This section presents an algorithm and its primary data structure for analyzing grammar decision points. The method is straightforward but has a higher complexity than the

fully implemented algorithm in our parser generator because computations are not saved for later reuse; computation caching is mentioned but not fully explored in this paper. Strong ***LL(k)*** grammars are considered here for simplicity, but PCCTS's analysis algorithm actually handles grammars that are between strong ***LL***(k) and ***LL(k)***; one can translate an ***LL(k)*** grammar to strong ***LL(k)*** [SiS82]. We also introduce the notion of ***FIRST*** trees as an efficient data structure for representing k-tuples.

*3.1. **FIRST*** k-Tuples vs. FIRST Trees

Representing a ***FIRST*** set as a set of k-tuples is convenient from a language theory point of view, but proves cumbersome when ***FIRST*** sets need to be built and manipulated by computer. Trees allow us to efficiently represent k-tuples during analysis and often highlight token comparison optimizations that are difficult to spot using tuples.

k-tuples can be represented in child-sibling tree form. For example, the tuples

$$(\alpha_1, \alpha_2, \alpha_3)$$
$$(\beta_1, \beta_2, \beta_3)$$
$$(\gamma_1, \gamma_2, \gamma_3)$$

can be represented by the following child-sibling tree:

$$\begin{array}{ccccc} \alpha_1 & \rightarrow & \beta_1 & \rightarrow & \gamma_1 \\ | & & | & & | \\ \alpha_2 & & \beta_2 & & \gamma_2 \\ | & & | & & | \\ \alpha_3 & & \beta_3 & & \gamma_3 \end{array}$$

or can be described using LISP-like notation as

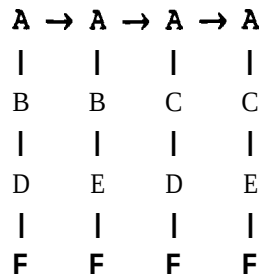
$$(\ \varnothing\ (\alpha_1\ (\alpha_2\ \alpha_3))\ (\beta_1\ (\beta_2\ \beta_3))\ (\gamma_1\ (\gamma_2\ \gamma_3))\ )$$

where $(\rho\ \sigma_1\ \sigma_2 \ldots a,)$ is a tree with p at the root and σ_i as the i^{th} child of p. The O in the root position is some nil node. ***All*** tokens at the same level represent the same token of lookahead; i.e. τ_i will match a token at the i^{th} level.

Because of the nature of grammars, ***FIRST*** trees contain many common subtrees; consequently many of the standard tree compactions associated with common subexpression elimination used in code generation technology can be applied. Also, many sequences have common token prefixes which can be factored out. The $FIRST_4$ tree for rule s from the grammar,

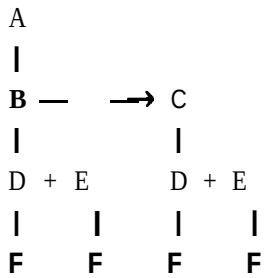
```
s  :  A t u F G ;

t  :  B | C ;

u  :  D I E ;
```

is represented by



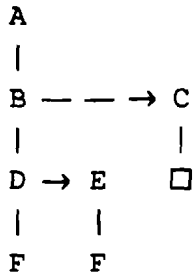
(T1a)

which can be factored to



(T1b)

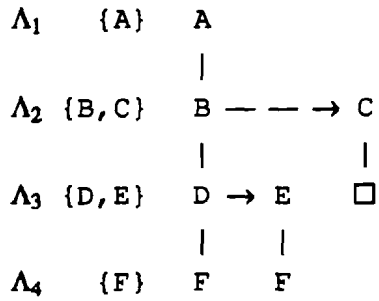
and can be further reduced by eliminating common subtrees to



(T1c)

where □ is a place holder representing the ((D F) (E F)) subtree tree we eliminated.

The Λ_i sets used in $LL_1(n)$ analysis can be calculated easily by collecting all tokens at level i in any of the trees (T1a), (T1b), (T1c); however, it is more efficient to traverse the reduced tree (T1c). For example, the Λ_i sets for rule s above are:



(T1d)

These tree structures are used extensively by the $LL(k)$ analysis algorithm outlined below and are an essential feature of our computation caching mechanism.

3.2. Algorithm Overview

A parser generator searching for an efficient expression to correctly predict alternatives examines templates of the form $LL_m(n)$ for some $m \leq n$ and $n \leq k$. $m=n=1$ is an obvious place to begin, choosing larger and larger m and n until a satisfactory template has been found. Our algorithm considers only m=1 and m=n ($m=0$ requires no decision template) as per section 2.3. For $m=1$, one does not require the tree structures of section 3.1 since A sets are the result. $FIRST_n(\omega)$ for m=1 returns a set of tokens that can be seen at 2, whereas $FIRST_n(\omega)$ for m=n returns a tree representing all possible input sequences generated by a $\omega \in V^* \cup N^*$. Therefore, two different algorithms may be used: one manipulating sets ($m=1$) and one manipulating trees ($m=n$). The set manipulating algorithm is substantially faster, but is less interesting than the m=n case.

A FIRST, request must reuse results from $FIRST_{n-1}$ in order to be efficient. In addition, If a FIRST computation requires the FIRST of another rule, this result must also be cached. Unfortunately, some rather devious programming is involved when cycles exist (when FIRST computations are mutually defined). The m=n FIRST, algorithm outlined in the next section ignores caching for the sake of clarity.

33. Basic Algorithm

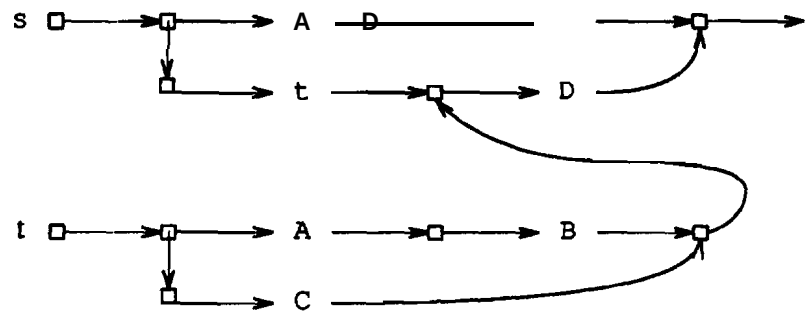
Our $FIRST_k$ algorithm constructs FIRST trees of depth k as outlined in section 3.1. The i^{th} level in the tree represents all tokens than can be matched by τ_i (the i^{th} lookahead token). As a byproduct, any $FIRST_k$ tree contains all FIRST, trees for $n \leq k$.

This section gives an extremely terse description of the data structure and algorithm needed to implement $FIRST_k$. Although we provide actual pseudo-C, it is incomplete in that it does not handle computations which are defined in a mutually recursive way (infinite recursion can occur). As mentioned previously, computation caching is an involved process and is also not incorporated here. An example FIRST computation is included.

A judicious choice of data structures for the grammar simplifies our algorithm considerably. In particular, we represent a grammar as a syntax diagram with special "FOLLOW-links" alleviating the need for a $FOLLOW_k$ function. We have effectively defined FOLLOW in terms of FIRST by modifying a standard syntax diagram to include links from the right edge: of all rules to the nodes; immediately following any reference to that rule. Interestingly, Extended BNF notation (EBNF) can be directly encoded as a slightly augmented syntax diagram without translating it to BNF beforehand; implying that our algorithm works identically for BNF and EBNF (well, almost). To illustrate our syntax diagram, consider the following grammar.

```
s   :  A D
    |  t D
    ;

t   :  A B
    |  C
    ;
```

The grammar can be represented by the following syntax diagram:



Note the link from the end of rule t to the junction node following the reference to t in rule s, production two. Any $FIRST_k$ computation that does not find complete k-tuples must compute a *FOLLOW* set. Rather than invoke another algorithm, we observe that a FOLLOW computation is nothing but a FIRST performed upon whatever follows all references to the rule with insufficient k-tuples. The link allows the *FIRST* computation to simply "fall of the edge" of the rule and pursue *FIRST*'s in other rules. When examining the algorithm given below, note that the junction blocks immediately following rule names are considered rule junctions and the junctions immediately preceding rule alternatives are considered alternative junctions.

The following pseudo-C code accepts a pointer to any position within the syntax diagram and returns a tree representing the sequences of tokens recognizable starting at that position.

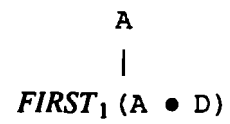
```
Tree *
FIRST_k(pos)
Node *pos;
{
    Tree *t_i;

    if ( pos->type == TERMINAL )
    {
        if ( k == 1 ) return mknode( pos->token );
        else return mktree( mknode(pos->token), FIRST_{k-1}(pos->next) );
    }
    if ( pos->type == RULE )
    {
        for ( i=1; i≤a; i++)

            t_i = FIRST_k( pos->rule->alt_i );
        }
        return mktree( NULL, t_1, t_2, ..., t_a );
    }
}
```

where mknode is a function that creates a Tree node from a token and mktree(p, σ_1, ..., σ_n) combines nodes/sub-trees to form a tree (p is the root, σ_i is the i^{th}

sibling as before). `pos->type` is the type of object found at node `pos` and `pos->rule->alt`$_i$ represents the i^{th} alternative of the rule referenced at `pos` where `pos->rule` points to the rule block of the rule referenced at `pos`.

Constructing a parser for rule `s` amounts to calling $FIRST_2$(s) which in turn computes $FIRST_2$ for productions `A D` and `t D`. Computation proceeds as follows (• indicates a position within a production):

$FIRST_2$(• A D) returns a tree comprised of



where $FIRST_1$(A t D) yields a node with `D` inside:

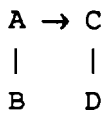
$FIRST_2$(• t D) returns a tree with two siblings:

$$FIRST_2(\bullet\ A\ B) \rightarrow FIRST_2(\bullet\ C)$$

$FIRST_2$(a A B) returns

and $FIRST_2$(• C) returns

$FIRST_1$(C •) traverses the FOLLOW-link pointing into production two of rule `s` and returns a node made from `D`. $FIRST_2$(a t D) eventually returns



A parser for rule `s` could now be constructed in the following way

```
s()
{
    if ( τ1 == A && τ2 == D ) {
        ...
    }
    else if ( (τ1 == A && τ2 == B) || (τ1 == C && τ2 == D) ) {
        ...
    }
}
```

The if expressions are not the fastest possible since, if one is guaranteed to have valid input, the second if expression is unnecessary. If the input were anything but A D, the parser would default to the second production.

This section presented an introduction to the problem of generating $LL(k)$ parsers with regards to $LL(k)$ grammar analysis. The actual algorithm has a great many more details whereas the algorithm given here is simple (and unrealistic). It does not handle non-strong $LL(k)$ grammars and, without caching, analysis time complexity can approach the upper bound established by [SiS83]: $O(|G|^{k+1})$ to test for the $LL(k)$ property.

3.4. Caching

A practical analysis algorithm must not compute any FIRST set more than once. Therefore, all computations must be cached for possible future retrieval. This can be accomplished by saving results in the junctions nodes (represented by small boxes) in the above syntax diagram. At most one full $FIRST_k$ tree is stored in each junction of the syntax diagram since FIRST, is contained in $FIRST_k$ for all $n \leq k$. Notice that this also implies that $FIRST_{n+1}$ may begin computation where FIRST, finished and, in fact, must do so to be efficient. FIRST trees are, therefore, augmented with "continuation" nodes at the leaves to indicate where in the syntax diagram the previous computation left off.

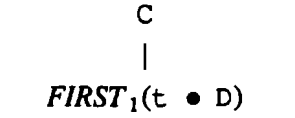
A FIRST, computation for some n may return immediately when it encounters a junction with a cache entry for FIRST,. For example, computing $FIRST_2(s)$ for the above grammar requires $FIRST_2(t)$:

```
A → FIRST2(t)
|
D
```

where $FIRST_2(t)$ is

```
A → C
|   |
D   D
```

since $FIRST_2(\bullet C)$ is



thanks to the FOLLOW link. When $FIRST_2(t)$ is needed to generate a parser for rule t, $FIRST_2$ can immediately return with the result since it was cached in the rule junction (first little block in the syntax diagram above) of t by the $FIRST_2(s)$ request.

This mechanism is essentially correct, but does not treat stores of partial computations — i.e. those that were started, but could not be immediately completed. These partial results must be saved because they often comprise much of the analysis run-time. Computations terminate early only when grammar cycles exist; primarily in situations where the $FOLLOW_k(a)$ set for some rule a indirectly or directly requires the FOLLOW of itself. A full discussion is not possible here, but caching is not as simple as computing a set and then storing it in a syntax diagram node. Refer to the on-line materials for PCCTS for further details.

3.5. Complexity Analysis

Analyzing an entire grammar to determine $LL(k)$-decidability has an upper bound of time $O(|G|^{k+1})$ and space $O(|G|)$ [SiS83]. We have developed an algorithm that has a time complexity of $O(|G| \times k)$ and a space complexity of $O(|G| \times |V|^k)$. The caching described above guarantees that we will compute FIRST, $(n \leq k)$ for any point in the grammar at most one time. If there are roughly $O(|G|)$ positions in a grammar and k different possible FIRST computations at each position, at most $O(|G| \times k)$ canonical set operations can be performed. The $|G| \times k$ FIRST sets cached in the syntax diagram by our algorithm each require space proportional to $|V|^k$, but $FIRST_k$ contains $FIRST_n$ for all $n \leq k$; this results in our $O(|G| \times |V|^k)$ space complexity. Because most parsing decisions require small amounts of lookahead, analysis time and space requirements are nearly always much lower than the worst case.

4. Example Using LL(k)

We have constructed a parser generator that accepts $LL(k)$ Extended BNF grammars. All previous sections dealt with a parser construction from a hypothetical point of view. This section presents an example grammar that is translated to C by the current version of PCCTS [PDC92]. The grammar is nonsensical but is simple and can be quickly grasped by the reader.

4.1. What PCCTS Implements

PCCTS is a set of public domain software tools designed to facilitate the implementation of compilers and other translation systems. It has a number of features that make it more useful than other compiler construction systems (e.g. we have integrated the specification of lexical and syntactic analysis). Here, we discuss only its parser generation ability.

ANTLR[6], the parser generator program within PCCTS, translates EBNF grammars

[6] ANother Tool for Language Recognition

(Extended BNF, similar to [CoS70]) directly to a syntax diagram like form similar to that of section 3.1. A code generator then walks the syntax diagram making quests to a *FIRST* set/tree algorithm in search of a the parameters for a parsing template. ANTLR parsers are not full $LL(k)$; yet they are more powerful than strong LL(k) parsers because the handling of subrules is strong $LL(k)$.[7]

43. Grammar

This section presents a grammar that contains $LL_0(0)$, $LL_1(1)$ and $LL_2(2)$ constructs. We give an analysis and the generated parser in the following sections.

```
s   :  A B
    |  A E
    |  t
    ;

t   :  A D
    |  C B
    |  C E
    ;
```

43. Analysis

ANTLR reports the following $FIRST_2$ trees for our grammar (using the LISP-like notation outlined above):

```
s   :  A B      /* (A B) */
    |  A E      /* (A E) */
    |  t        /* ( (A D) (C B E) ) */
    ;

t   :  A D      /* A */
    |  C B      /* (C B) */
    |  C E      /* (C E) */
```

Notice that ANTLR did not consider two tokens of lookahead for the first production of rule t because the choice is uniquely determined by τ_1.

4.4. Generated Parser

The following C code was generated by ANTLR for rules s and t above (minus the attribute and error handling code).

[7] Full LL(k) parsing for PCCTS is currently under development.

```
s()
{
    if ( (LA(1)==A) 66 (LA(2)==B) ) {
        match(A); CONSUME;
        match(B); CONSUME;
    }
    else if ( (LA(1)==C) 66 (LA(2)==D) ) {
        match(C); CONSUME;
        match(D); CONSUME;
    }
    else if ( (setwd[LA(1)]&0x1) 66 (setwd[LA(2)]&0x2) ) (
        t();
    }
}


t()
{
    if ( (LA(1)==A) ) {
        match(A); CONSUME;
        match(D); CONSUME;
    }
    else if ( (LA(1)==C) && (LA(2)==B) ) {
        match(C); CONSUME;
        match(B); CONSUME;
    }
    else if ( (LA(1)==C) 6h (LA(2)==E) ) (
        match(C); CONSUME;
        match(E); CONSUME;
    }
}
```

Here, `LA(i)` is equivalent to τ_i in our previous notation (the i^{th} token of lookahead). `match(τ)` verifies that τ is the current token and CONSUME fetches the next token of lookahead.

The decision for the third alternative in rule **s deserves** special attention. One would have expected a test of the form:

```
if ( (LA(1)==A || LA(1)==C) &&
     (LA(2)==B || LA(2)==D || LA(2)==E) ) {
    ...
}
```

Instead, ANTLR noted that only single-token comparisons were being made and a set operation would be faster than two comparisons to `LA(1)` or three comparisons to `LA(2)`. Each unique token set is expressed by a particular bit position within an array indexed by token. `setwd[LA(1)]&0x1` looks up the position in the `setwd` array of `LA(1)` and checks the first bit. If the bit is one, `LA(1)` is a member of that set. This operation is best described by the diagram

setwd	*set bits*
A	[0001]
B	[0010]
C	[0001]
D	[0010]
E	[0010]

which says that A and C are members of the first set (bit position 0; hence the &'ing with 0x01) and B, D and E are members of the second set (hence we mask with 0x02). This type of membership operation is $O(1)$, which results in much better recognition speed than the equivalent series of token wmparisons.

Likewise, it is important to note that the definitions of the C operators || (logical or) and && (logical and) result in some optimization of the sequential wmparisons. C evaluates these logical expressions left to right, but as soon as the result is known, the remainder of the expression is skipped. For example, if τ_1 were C, production one of rule s,

```
if ( (LA(1)==A) && (LA(2)==B) ) { ... }
```

would not bother testing LA(2) (τ_2) against B since the first subexpression is false [ANS90]. Execution would proceed immediately to the second alternative.

It is interesting to note that the type of code generated by ANTLR is sensitive to the ordering of alternatives. If rule s were rewritten to have the production referencing t as the first production, a different parsing template would be required for that production.

```
s   : t
    | A B
    | A E
    ;

t   : A D
    | C B
    | C E
    ;
```

Rule s, production one, is now $LL_2(2)$. As in section 2.3 we choose to augment the $LL_1(2)$ solution with a test that disallows the artificially created token sequences that clash with productions two and three (A B and A E). Specifically, ANTLR generates the following if statement to predict the production, which references t. in its new position.

```
if ( (setudl[LA(1)]&0x1) && (setudl[LA(2)]&0x2) &&
     !(LA(1)==A && (LA(2)==B || LA(2)==E)) ) {
    t();
}
```

Notice that the comparison for the tree

```
A → A
|   |
B   E
```

was left-factored implicitly to test τ_1 only once for A since the tree was reduced by the analysis algorithm to:

```
A
|
B → E
```

This section gave a quick demonstration of how the **PCCTS** parser generator, **ANTLR,** uses different $LL_m(n)$ parsing templates to generate efficient $LL(k)$ parsers. It also provided some insight into a few implementation issues like the fast set membership operation **and** how choice of language **(e.g.** the C language) effects recognition speed. However, we encourage the reader to further examine **PCCTS** to better understand the new techniques described here; **PCCTS** is public domain software and is available via electronic mail to `pccts@ecn.purdue.edu`.

5. Conclusion

$LR(k)$ parsers can recognize a larger class of languages than LL(k) parsers and can always be rewritten to **use** only one token of **lookahead —** greatly simplifying the parser. Although LL parsers are not as strong and typically require more lookahead than LR to recognize the same language, LL has many advantages over LR when translation versus simple recognition is the goal. LL parsers allow arbitrary action placement, downward inheritance and are significantly easier to **debug.**

Previously, the only LL parsers in common use were $LL(1)$, because k token lookahead was considered impractical. By observing that most LL parsing decisions in a typical grammar can be made without comparing k-tuples and often do not even require the full k tokens of lookahead, we **formulated** $LL_m(n)$ as a method of generating efficient LL (k) parsers. Further, we explained how this new **technique** has been implemented in the PCCI'S parser generator.

In summary, when building a translator for a language, LL**'s** superior **attribute/action** handling abilities make it preferable to LR. The primary contribution of this paper, $LL_m(n)$, makes $LL(k)$ efficient enough to be practical for nearly all translation problems involving source languages that have LL grammars.

6. References

[AhU72] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling Volume I, Prentice-Hall, 1972.

[AhU79] A.V. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[ANS90] American National Standard for Information Systems, "Programming Language C," ANSI Document Number X3J11/90-013, February 1990.

[CoS70] John Cocke and J.T. Schwartz, "Programming Languages and Their Compilers", Preliminary Notes, Second Revised Version, April 1970, Courant Institute of Mathematical Sciences; New York University.

[DoP90] H. Dobler and K. Pirklbauer, "Coco-2 A New Compiler Compiler," ACM SIGPLAN Notices, Vol. 25, No. 5, May 1990.

[FiL88] Charles N. Fischer and Richard J. LeBlanc, Crafting a Compiler. Benjamin/Cummings Publishing Company, 1988.

[Joh78] Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, Murray Hill, NJ, 1978.

[JuD84] R. Juels, H. Dietz, S. Arbeeny, J. Patane, E. Tepe, "Automatic Translation Systems," Study Report, Center For Digital Systems, Department of Electrical Engineering and Computer Science, The Polytechnic Institute of New York, New York, 1984.

[Knu71] Donald E. Knuth, "Top-Down Syntax Analysis," Acta Informatica 1, 79-110, 1971.

[LeS68] P.M. Lewis II and R.E. Stearns, "Syntax-Directed Transduction," Journal of the ACM, Vol 15, No. 3, 1968, pp 465-488.

[MiF79] D.R. Milton and C.N. Fischer, "LL(k) Parsing for Attributed Grammars," Automata, Languages and Programming, Sixth Colloquium, 1979, pp 422-430.

[NiP82] Anton Nijholt and Jan Pittl, "A General Scheme for Some Deterministically Parsable Grammars and Their Strong Equivalents," (Extended Abstract), Lecture Notes in Computer Science 145, Springer, Berlin, 1982; pp 243-255.

[Par90] T.J. Parr, "The Analysis of Extended BNF Grammars and the Construction of LL(1) Parsers," M.S. Thesis, Purdue University School of Electrical Engineering, May 1990.

[PDC92] T.J. Parr, H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual Version 1.00," *to appear* in *ACM SIGPLAN Notices early 1992.*

[PDC92a] T.J. Parr, H.G. Dietz, and W.E. Cohen, "On k-Token Lookahead and the Equivalence of *LL* and *LR* Translators," *submitted to ACM PLDI 1992.*

[PuC89] James J. Purtilo and John R. Callahan, "Parse-Tree Annotations", Communications of the ACM. Vol. 32, No. 12, December 1989.

[Rob90] George H. Roberts, "From Recursive Ascent to Recursive Descent: Via Compiler Optimizations," SIGPLAN Notices, Vol. 25, No. 4, April 1990.

[SiS82] Seppo Sippu and Eljas Soisalon-Soininen, "On L L Q Parsing," Journal of Information and Control, Vol 53, 1982. pp 141-164.

[SiS83] Seppo Sippu and Eljas Soisalon-Soininen, "On the Complexity of LL(k) Testing," Journal of Computer and System Sciences, Vol 26, 1983. pp 244-268.

[Ukk83] Esko Ukkonen, "Lower Bounds on the Size of Deterministic Parsers," Journal of Computer and System Sciences, Vol 26, 1983. pp 153-170.

Table of Contents