Tools of Computing Technical Report 1999-3-#2-TC, 3/12/1999

A Strong LL(k) Parser Generator That Accepts Non-LL Grammars and Generates LL(1) Tables

Thomas Christopher
DePaul University, CTI
243 S. Wabash Ave.
Chicago IL 60604
tc@toolsofcomputing.com

Abstract

We have implemented a strong LL(k) parser generator, TCLLk, with three major innovations: (a) The parser generator rewrites the grammar to put it in close to LL(1) form. (b) If look-aheads of more than one symbol are needed, it builds look-ahead trees out of LL(1) productions. (c) It uses a novel computation of $FOLLOW_k$ sets.

We compare its performance to LALR(1) for several grammars and report our experience translating a Java grammar into a form suitable for TCLLk. The algorithms in TCLLk appear to offer a significant improvement over LALR(1).

Keywords: LL(k) parsing

Introduction

LL(k) [Lewis68] [Rosenkrantz70] and LR(k) [Knuth65] parsers have become dominant for compiler construction. Indeed, concerns for the size of the parsing tables have restricted the choice further to LL(1) and LALR(1) [DeRemer69] [DeRemer82]. LL(1) is credited with having smaller parsing tables and better error recovery, whereas LALR(1) accepts a larger class of grammars and hence does not require as much expertise on the part of the compiler writer. Indeed, the difficulty of converting programming language grammars to LL(1) form is the biggest argument against LL(1). Due to its ease of use, LALR(1) is usually preferred.

We have developed a strong LL(k) parser generator, TCLLk, that promises to tilt the preference back to LL parsers. Our parser generator transforms programming language grammars into a form close to LL(1) by removing left recursion from grammars and factoring. The class of grammars accepted appears to be about as convenient for compiler writers as that accepted by LALR(1). Both TCLLk and LALR(1) can accept grammars the other can't.

We provide k-symbol look-ahead where needed by building look-ahead trees. As Parr [Parr93] points out, k-symbol look-ahead has been rejected for requiring space exponential in k, but that is based on the assumption that one must use k-tuples of look-ahead symbols. In practice, by handling the look-aheads one symbol at a time, the space required for look-ahead is quite modest. We differ from Parr in how we compute and represent the look-aheads, and instead of having heterogeneous parser states, our parser uses LL(1) tables.

The size of parse tables are shown to be smaller than those that would be generated from LALR(1).

Preliminaries

Let T be a set of symbols. T* is the set of all strings composed of symbols taken from set T, including the empty string, ε. The length of a string u is written |u|.

A context-free grammar (CFG) is a four-tuple (N,T,s,P) where N is a set of nonterminal symbols, T is a set of terminal symbols, $s \in N$ is the start symbol, and P is a set of productions, $A \rightarrow u$, where $A \in N$ and $u \in (N \cup T)^*$. $A \rightarrow x|y|...|z$ is short for $A \rightarrow x$, $A \rightarrow y$, ..., $A \rightarrow z$. The productions represent rewriting rules. Given a production $A \rightarrow u$, the string xAy may be rewritten xuy, expressed as $xAy \Rightarrow xuy$. A series of zero or more rewritings, $u \Rightarrow^* v$, is called a derivation. A derivation of one or more rewritings is denoted

$u \Rightarrow^+ v$.

A context free grammar with action symbols, CFGA, is a five tuple (N,T,𝒜,s,P) where 𝒜 is a set of action symbols, N, T, and s are as in CFGs, and P is a set of productions, $A \rightarrow u$, where $A \in N$ and $u \in (N \cup 𝒜 \cup T)^*$. Action symbols are used to interface a parser to the semantics in a compiler. As terminal symbols are recognized, they are pushed on a semantics stack. When the parser encounters an action symbol, it calls an action routine which removes several values from the semantics stack and may push a value back on the stack. The actions symbols are typically placed at the ends of productions to synthesize the value of the left-hand-side symbol from the values of the right hand side symbols. Action symbols are not present in the input language. One may describe that language with an *underlying CFG*: $(N \cup 𝒜, T, s, P \cup \{A \rightarrow \varepsilon \mid A \in 𝒜\})$. That is to say, action symbols behave like nonterminals that derive only the empty string.

The function $FIRST(x) = \{a \mid x \Rightarrow^* av\}$ where the derivation is done in the underlying CFG. The functions $FIRST_k(x) = \{u \mid u \in T^*$ and either $x \Rightarrow^* u$, $|u| \leq k$ or $x \Rightarrow^* uv$ and $|u| = k\}$. Again, the derivation is done in the underlying CFG. Note that FIRST(x) includes all types of symbol, but $FIRST_k$ only includes strings of terminals. $FIRST_k$ applied to a set of strings is the union of $FIRST_k$ applied to each string in the set. The k symbol look-ahead of a production $A \rightarrow u$, k-look-ahead($A \rightarrow u$), is $FIRST_k(\{uy \mid s \Rightarrow^* xAy \Rightarrow xuy\})$. A grammar is strong LL(k) (SLL(k)) if for all productions $A \rightarrow u$ and $A \rightarrow v$ in P,

k-look-ahead($A \rightarrow u$)$\cap$k-look-ahead($A \rightarrow v$)=$\varnothing$

The LL(1) parsing algorithm is given in Figure 1. It works by generating a sentence and matching it to the input as it is generated. The prediction stack, also known as a parse stack, holds the right part of the sentence being generated. This version of an LL(1) parser is designed for interactive use; it can perform a series of actions before reading the next input symbol.

An operation we use extensively in our algorithm is expanding a nonterminal in a production. Suppose we have a production $A \rightarrow uBv$ and the entire set of productions for B are $B \rightarrow x_1$, $B \rightarrow x_2$, ..., $B \rightarrow x_n$. To expand B in the production $A \rightarrow uBv$, we replace $A \rightarrow uBv$ with productions $A \rightarrow ux_1v$, $A \rightarrow ux_2v$, ..., $A \rightarrow ux_nv$.

Method

Our algorithm tries to produce LL(1) parsing tables for a CFGA by first rewriting the grammar in an attempt to put it in LL(1) form, and then, if the rewriting is not completely successful, by building look-ahead trees. More specifically, the parser generator will:

1. Analyze the grammar and stop if it is not reduced.
2. Remove left recursion.
3. Factor.
4. Build look-ahead trees, if necessary.
5. Write out LL(1) parse tables.

Left recursion removal. A production $A \rightarrow Au$ is directly left recursive; nonterminal A can derive itself first in a string by one rewrite. More generally, a nonterminal A is left recursive if $A \Rightarrow^+ Av$ in the underlying grammar. LL parsers cannot be produced for grammars containing left recursive nonterminals.

The fundamental transformation of left recursion removal is to remove direct left recursion:

$$A \rightarrow A\,u_1 \mid A\,u_2 \mid \ldots \mid A\,u_n \mid w_1 \mid \ldots \mid w_m$$

becomes

$$A \rightarrow w_1\,A' \mid \ldots \mid w_m\,A'$$

$$A' \rightarrow u_1\,A' \mid u_2\,A' \mid \ldots \mid u_n\,A' \mid \varepsilon$$

where A' is a new nonterminal.

A production $A \rightarrow u$ is indirectly left recursive if it is not directly left recursive and $A \in FIRST(u)$. Indirect left recursion can sometimes be converted into direct left recursion and eliminated as shown above. There are two cases:

Case 1.

$A \rightarrow Bv$, $B \in N$, $B \neq A$

Expand B in $A \rightarrow Bv$, and attempt to remove left recursion from the new productions introduced.

Case 2.

$A \rightarrow bv$, $b \in \not\Delta$

Since b is an action symbol, we cannot rewrite it, we cannot remove left recursion, and hence we cannot produce an LL parser for the grammar.

Factoring. The factoring phase attempts to rewrite the grammar so that for any two productions for the same nonterminal, $A \rightarrow u$ and $A \rightarrow v$,

$FIRST(u) \cap FIRST(v) = \varnothing$.

The trivial way for $FIRST(u) \cap FIRST(v) \neq \varnothing$ is for u and v to begin with the same string of symbols, y. These may be directly factored as follows:

$A \rightarrow y\ u_1 \mid ... \mid y\ u_n \mid w_1 \mid ... \mid w_m$

becomes

$A \rightarrow y\ A' \mid w_1 \mid ... \mid w_m$

$A' \rightarrow u_1 \mid ... \mid u_n$

where A' is a new nonterminal symbol. Direct factoring consolidates several productions that begin with the same sequence of symbols.

However, more than one right hand side may derive the same symbol(s) first without having a common prefix. In this case the parser generator resorts to *deep factoring*: expanding initial nonterminals in right hand sides until direct factoring is possible.

To decide which nonterminals to expand, the parser generator computes the conflict set, C for a nonterminal A with productions $A \rightarrow u_1 \mid ... \mid u_n$:

$$C = \bigcup_{i \neq j} (FIRST(u_i) \cap FIRST(u_j))$$

which is to say, C is the set of symbols that are in more than one FIRST set.

Deep factoring expands nonterminal B in production $A \rightarrow Bv$ if $FIRST(Bv) \cap C \neq \varnothing$ and $B \notin C$. Replacing B with each of its right hand sides will eventually produce productions that begin with the same strings of symbols and can be directly factored. If B is in the conflict set, we do not expand it because B itself is a symbol that can be factored. The process will terminate because left recursion has already been eliminated.

Direct factoring, as shown above, introduces a new nonterminal and productions. Those productions may require deep factoring, leading to further deep factoring, and so on. To avoid infinitely many rewritings, the parser generator limits the number of levels it will pursue this process.

At this point, the grammar may already be in LL(1) form. If so, the parser generator need not generate any look-ahead trees.

Generating look-ahead trees. The parser generator calculates the follow set, follow(A), for each nonterminal A, the set of terminal symbols that can follow A in some sentential form.

$follow(A) = \{ t \mid t \in T \text{ and } s \Rightarrow^* uAtv \}$

If A has productions $A \rightarrow w \mid u_1 \mid ... \mid u_n$ where $w \Rightarrow^* \varepsilon$, then the grammar is not LL(1) if

$FIRST(u_i) \cap follow(A) \neq \varnothing$

for any of the productions $A \rightarrow u_i$.

The parser generator must decide whether to build a look-ahead tree or to treat A as a *dangling-tail* (a generalization of the dangling-else, which cannot be made LL(k)).

The parser generator considers a nonterminal A to be a dangling tail if it derives the empty string and its only occurrences on right hand sides are of one of the two forms:

$B \rightarrow u\ D\ A$

where $D \in N$, and $D \Rightarrow^* vA$, or

$B \rightarrow A$

and B is a dangling tail.

Dangling tails are handled in the usual fashion; the parser is instructed to prefer a non-empty-deriving alternative to the empty-deriving one.

If the productions for a nonterminal are neither LL(1) nor an instance of a dangling tail, the parser generator will try a look-ahead of up to k symbols. The reason SLL(k) grammars have been rejected for k>1 is that they were formulated in terms of k-tuples of look-ahead symbols, which requires excessive table space, $O(|N|^k)$. Our parser generator uses look-ahead trees, and that only where necessary, usually resulting in only modest space requirements.

The look-ahead tree is represented as a finite state automaton (fsa) whose transitions, in turn, are represented as a right regular grammar. Each fsa state is represented by a new *look-ahead* nonterminal. Let the set of look-ahead nonterminals be $\mathcal{L}$. The fsa transitions are represented by productions

$A \rightarrow tB$

where $A \in \mathcal{L}$, $B \in \mathcal{L}$, and $t \in T$. The accepting states are represented by productions

$D \rightarrow h^j u$

where $h \in \mathcal{A}$ is a special action symbol and the fsa has found that it should replace a nonterminal, E, with string u (i.e. production $E \rightarrow u$) after a look-ahead of $j \leq k$ symbols. The action h removes one token from the semantics stack and pushes it back in front of the input to be read again later.

The computation of the look-ahead tree is based on a novel computation of the $FOLLOW_k$ relation.

$FOLLOW_k(A)=\{x \mid s\$^k \Rightarrow^* uAv\$^k$ and $x \in FIRST_k(v\$^k)\}$

where $\$^k$ represents a string of k end-of-input symbols.

The parser generator creates a set, $\mathcal{J}$, of follow nonterminals, F_A, one for each nonterminal A of the CFGA, and a set of follow productions, $\mathcal{P}$. For each occurrence of a nonterminal A on the right hand side of a production $B \rightarrow uAv$ in the CFGA, the parser generator creates a follow production

$F_A \rightarrow v\ F_B$

For the start symbol, it creates a production

$F_s \rightarrow \$^k$

Recalling the underlying CFG is

$G=(N \cup \mathcal{A}, T, s, P \cup \{A \rightarrow \varepsilon \mid A \in \mathcal{A}\})$

observe that CFG

$(\mathcal{J}, N \cup T \cup \mathcal{A}, F_A, \mathcal{P})$

is a grammar for the symbols that may be on the prediction stack beneath an A. Similarly, CFG

$G'=(\mathcal{J} \cup N \cup \mathcal{A}, T, F_A, P \cup \mathcal{P} \cup \{A \rightarrow \varepsilon \mid A \in \mathcal{A}\})$

describes the strings of terminal symbols that may follow an A in a sentential form. Finally we observe that

$FOLLOW_k(A)=FIRST_k(F_A)$

where $FOLLOW_k(A)$ is computed in grammar G and $FIRST_k(F_A)$ is computed in grammar G'.

Beginning look-ahead. If A has productions $A \rightarrow w \mid u_1 \mid ... \mid u_n \mid v_1 \mid ... \mid v_m$ where $w \Rightarrow^* \varepsilon$, $FIRST(u_i) \cap follow(A) \neq \varnothing$ and $FIRST(v_j) \cap follow(A) = \varnothing$, build a list of states, SL. Each state has two components (r,p), where r is a *prediction string* and p, a *production*. Create the following states and insert then into SL:

$(w\ F_A, A \rightarrow w)$, where $w \Rightarrow^* \varepsilon$

$(u_i\ F_A, A \rightarrow u_i)$, where $FIRST(u_i) \cap follow(A) \neq \varnothing$

This list of states is passed to a procedure buildLookAheadTree which returns a nonterminal. Let

B=buildLookAheadTree(SL,0)

We replace the productions for A with $A \rightarrow B \mid v_1 \mid ... \mid v_m$.

Algorithm for buildLookAheadTree(SL,d). SL is a list of states and d is the depth, an integer.

Step 1. If all the states in SL have the same production, $A \rightarrow u$, create a new nonterminal B and a production

$B \rightarrow h^d u$

where h is the action symbol to backup, i.e. pop the top token off the semantics stack and push it back in front of the input. Return B.

Step 2. If d is greater than the maximum look-ahead depth, k, then report that a look-ahead greater than k symbols is required.

Step 3. Process the list of states, SL, as follows until each prediction string begins with a terminal symbol:

Case 1: (a u, p), where a∈Σ

If the prediction string begins with an action symbol, remove (a u, p) and insert (u,p) in its place.

Case 2: (B u, p), where B∈N

Expand B, i.e. remove (B u, p) and insert (vu,p) for each production B→v.

Case 3: (F_B, p)

Remove (F_B, p) and insert (v, p) for each follow production F_B→v, unless F_B has already been expanded in the current set of states. Suppose there are two states (F_B, p) and (F_B, q). If p≠q then there will be no string of look-ahead symbols to distinguish between p and q. If p=q then the insertion of the states (v, p) would be redundant and could result in an infinite loop in the parser generator.

Step 4. Once all the states begin with terminal symbols, create a new look-ahead nonterminal B. Partition the states into sublists whose prediction strings all begin with the same symbol. Let L be the partition where all predictions begin with t, and L' be L where the initial t removed from each prediction. Let

C=buildLookAheadTree(L',d+1)

Add a look-ahead production

B→t C

When all the partitions have been processed, return B.

Results

The current version of the SLL(k) parser generator, TCLLk [Christopher99], is written in Icon. It incorporates panic mode error repair.

There are a number of questions to be answered about a new parser generation algorithm, the current favorite parsing algorithm:

- What is the class of grammars accepted? Or more importantly, how convenient is it for expressing programming language grammars?
- What is the size of the parsing tables produced?
- What is the speed of the parsers?

We are particularly interested in the answers in comparison to LALR(1), the currently most popular parsing algorithm.

Generality. To show that neither the class of grammars accepted by LALR(1) nor that accepted by our rewriting SLL(k) technique includes the other, we experimented with two grammars:

The first was a simple grammar known not to be LL(k) for any k, but which is LALR(1). As expected, TCLLk could not generate a parser for this grammar.

The second was a grammar which is LR(1), but not LALR(1). TCLLk successfully generated a parser for this grammar.

Table size. Estimates are that LL(1) produces parsing tables about one third the size of LALR(1) tables for the same size grammar as grammars grow larger [Fischer88]. For the same programming language, however, the LL(1) grammar is larger than the LALR(1) grammar. To see what effect TCLLk would have on the sizes of parse tables, we applied it to several programming language grammars.

When applied to the sample grammars, the number of nonterminals, productions, and the total number of symbols on the right hand sides of productions grew as shown in Table 1.

With the exception or Java, there are no expansions of more than a factor of 2 in numbers of nonterminals, productions, or summed lengths of productions for and language.

To compare to LALR(1), we passed the grammars through a translation program to put them in Yacc form and then passed them through Bison. Table 2 shows the number of shifts, reduces, states, and con-

flicts Bison reported. Only the EULER grammar was completely acceptable. The grammars with shift/reduce conflicts may produce correct parsers; they can be caused by such things as dangling elses, and they are resolved by shifting. The reduce/reduce conflict for the C grammar guarantees that it's LALR(1) parser isn't correct. The problems do not prevent us from using the grammars to compare table sizes.

Figure 2. shows a comparison of parse table sizes. The LALR(1) sizes are computed using Fischer's and LeBlanc's formula [Fischer88], from the parsers produced by Bison. It assumes the empty entries are compressed from the tables, but does not consider other size optimizations. The TCLLk are estimated based on the fraction of selection table occupied, the sum of lengths of the right hand sides, and the number of elements in the default table (explained next). It omits the error recovery information. The units are not bytes. They are for crude comparison purposes only.

The *default table* in the TCLLk is used as follows: The parser looks up the nonterminal on the top of the prediction stack and the next symbol in the input in the selection table. If it finds an associated right hand side, it replaces the nonterminal with that right hand side. If it doesn't find the pair, then it looks up the nonterminal in the default table. If the default table maps the nonterminal into a right hand side, it replaces the nonterminal with that. If neither table specifies a right hand side, the parser has detected an error in the input.

TCLLk always uses the default table for nonterminals that have only a single right hand side. That is all it uses the table for normally. The mode with *aggressive defaulting* also includes in the default table the right hand side selected by the most terminals, omitting it from the selection table. The saving in space is evident from Figure 2. TCLLk's parsers without aggressive defaulting are noticeable smaller than the LALR(1) parsers without compression. TCLLk's parsers with aggressive defaulting can be expected to be *much* smaller than LALR(1) parsers that were compressed by removal of error entries, but not by such things as merging rows of the action table. This leads us to expect we will find them still to be smaller than LALR(1) with more ambitious compression.

A Java Grammar. To test TCLLk against LALR(1) for a practical language, we cut and pasted an LALR(1) Java 1.1 grammar from The Java Language Specification [Gosling96] and reworked it to be acceptable to TCLLk. The authors of the Specification discuss the problems they had in converting their Java grammar to LALR(1) form. Here we discuss the difficulties we had converting their LALR(1) grammar for TCLLk's use.

First, we passed the grammar through a small Icon program to put it in proper input syntax for TCLLk. TCLLk then reported nine errors. These errors came from three sources.

First, Java has a dangling else clause. LL parser generators have to give dangling elses special treatment. LR parser generators can handle them in the grammar. The Java grammar included a number of nonterminals with names containing "NoShortIf", e.g. StatementNoShortIf. Short ifs are if-statements without else-clauses. The rule is that an if statement with an else clause cannot contain an if statement without an else before its else. We removed the "`NoShortIf`" versions of statements, a simplification of the grammar that removed nine productions.

Second, TCLLk complained about the number of factorings needed and about look-ahead depth being exceeded. The complaints involved various levels of expressions.

To find the problem, we started with Primary and related nonterminals and repeatedly added more expressions to it, running it through TCLLk, looking for where the problems occurred. The errors occurred when AssignmentExpression was added.

One of the nasty constructs for LL parsers is assignment expressions. They are high level expressions—many levels from identifiers and literals—but their left hand sides are usually some low level of expression. Handling them requires deep factoring, replacing nonterminals with their right hand sides repeatedly until direct factoring is possible.

TCLLk tried deep factoring, but for whatever reason, it didn't work. We resorted to a trick often used when building parsers: we replaced the left hand side with a slightly higher level of expression, the lowest level at which TCLLk reported no errors, knowing that this will accept syntactically illegal constructs, but that we can reject them in the semantics routines.

Third and finally, there was a problem with switch statements. They have an optional list of SwitchBlockStatementGroups followed by an optional list of

SwitchLabels just before the final "}".

```
SwitchStatement =  switch "(" Ex-
   pression ")" SwitchBlock .
SwitchBlock =  "{" SwitchBlock-
   StatementGroupsopt
   SwitchLabelsopt "}" .
```

Since each SwitchBlockStatementGroup begins with one or more SwitchLabels, there was a look-ahead conflict: seeing "case", a parser couldn't decide for SwitchBlockStatementGroupsOpt whether to look for SwitchBlockStatementGroups or choose the empty alternative.

We used TCLLk's extended input syntax to solve the problem:

```
SwitchBlock =  "{"
   { SwitchBlockStatementGroup }
   SwitchLabelsopt "}" .
```

The repetitive form, `{x}`, means zero or more repetitions of x. TCLLk translates

```
A = u { v } w.
```

into

```
A = u A'.
A' = w .
A' = v A'.
```

This allows the w and v to be factored. In the switch statement, it allows the final optional SwitchLabel to be factored against the SwitchLabel that begins a SwitchBlockStatementGroup. This worked, and TCLLk found no further problems with the Java grammar.

Overall, there was not much work in converting an LALR(1) grammar to TCLLk using the default limit of 3 factorings per nonterminal and a 2 symbol look-ahead.

Table 3 shows the figures for the Java grammar before and after these by-hand transformations and after TCLLk processed it.

How long does TCLLk take to build and write out parse tables for Java? The time taken to load and run TCLLk on the Java grammar was about 8 seconds on a 200 MHz Pentium PC running Windows NT. A more careful experiment could be done easily, but it doesn't seem worth it: The execution time is trivial.

Parser speed. Both LALR(1) and LL(1) parsers are linear time in the length of the input. How will k-symbol look-ahead affect TCLLk's parser's speed?

Look-ahead won't change the linear time. A full k-symbol look-ahead reads k symbols then backs up k symbols, then reads one. Suppose k symbol look-ahead were required for every symbol the parser reads. That would increase the time it spends reading the program by a factor of (2k+1). Of course, the increase in compiler execution time is unlikely to be anywhere near that, because

- a compiler does more than read and recognize the program, so it's only a fraction that will take longer,
- practical programming language grammars should not have a k-symbol look-ahead on every symbol—indeed very few, and
- TCLLk can remove back-ups on look-ahead if the back-up action symbols are followed by terminal symbols.

To see what fraction of the tokens read might be backed up over and read again, we implemented a Java scanner and parser and tried it out on several Java code files. Table 4 shows the number of tokens read and backups.

For a parse using the Java grammar, about one backup occurred for each five tokens read. This was astonishingly high. An examination of the translated grammar showed the reason: a Name looked ahead beyond the Identifier. The cause appeared to be TypeImportOnDemandDeclaration:

```
TypeImportOnDemandDeclaration =
   import Name "." "*" ";".
```

which says that a Name can be followed by `". *"`. Since a Name can be a `"Name . Identifier"`, a "." following a Name required look-ahead.

The obvious optimization was to redefine ImportDeclarations as follows:

```
ImportDeclaration =  import Im-
   portName ";" .
ImportName = Identifier "." Im-
   portName.
ImportName = Identifier "." "*".
ImportName = Identifier .
```

After making this change, the backups are as shown in Table 4. The fraction of backups was made utterly trivial.

Discussion

We have presented a strong LL(k) parser generator that

- requires about as little manipulation of grammars by the user as LALR(1). It allows left recursion and intersecting First sets.
- produces parse tables highly competitive with LALR(1) tables in size.
- builds look-ahead trees when, and only when, greater than one-symbol look-ahead is required.
- uses LL(1) parsing tables.

Overall, we expect this technique to be highly competitive with LALR(1).

References

[Christopher99] Christopher, Thomas W., *User Manual for TCLLk: A Strong LL(k) Parser Generator and Parser*, Technical Report 1999-3-#1-TC, Tools of Computing LLC, P. O. Box 6335, Evanston IL, 60204-6335, http://www.toolsofcomputing.com, 1999.

[DeRemer69] DeRemer, F., *Practical translators for LR(k) languages.* Ph. D. Dissertation, MIT, 1969.

[DeRemer82] DeRemer, F., and T. Pennello, "Efficient computation of LALR(1) look-ahead sets." *ACM TOPLAS* **4**,4 (Oct. 1982), 615-649.

[Fischer88] Fischer, Charles N., and Richard J. LeBlanc, Jr., *Crafting a Compiler*, Section 6.11 "LL(1) or LALR(1), That Is the Question," The Benjamin/Cummings Publishing Company, 1988.

[Gosling96] Gosling, James, Bill Joy, Guy Steele, *The Java Language Specification*, Edition 1.0, Addison Wesley, 1996. HTML downloaded from www.javasoft.com.

[Knuth65] Knuth, D. E. "On the translation of languages from left to right." *Information and Control* **8** (1965), 607-639.

[Lewis68] Lewis, P. M. II, and R. E. Stearns, "Syntax-directed transduction," *JACM* **15** (1968), 464-88.

[Parr93] Parr, T. J., *Obtaining practical variants of LL(k) and LR(k) for k>1 by splitting the atomic k-tuple*. Ph. D. Dissertation, Purdue University, 1993.

[Rosenkrantz70] Rosenkrantz, D. J., and R. E. Stearns, "Properties of deterministic top-down grammars." *Information and Control* **17** (1970), 226-256.

Figure 1. LL(1) parsing algorithm.

Initially, place the start symbol and the EOI (end of input) symbol on the prediction stack with the start symbol on top. Put EOI at the end of the input. Make the current token empty. Make the semantics stack empty.

Repeat

Pop the *top symbol* off the prediction stack.

While it is an action symbol, call its action routine and pop the next *top symbol* off the prediction stack. The action routine may pop zero or more values off the semantics stack and may push one or zero values back on it.

If the current token is empty, call the scanner to read the next input token into the current token.

If the *top symbol* from the prediction stack is a terminal, compare it to the *current token*.

If they match, push the *current token* onto the semantics stack. Make the current token empty.

If they don't match, an error has been discovered in the input. Execute error recovery code.

Otherwise if the top symbol from the prediction stack is a nonterminal, then choose one of its right hand sides and push it on the prediction stack, rightmost symbol on bottom. Choose the right hand side by looking at the *current token* and deciding which right hand side will allow parsing to continue.

until the EOI symbol is matched.

Table 1: Increase in grammar size.

Grammar		Initial	After TCLLk	Growth
C	Nonterminals	80	133	66.3%
	Productions	215	323	50.2%
	Total RHS	401	599	49.4%
EULER	Nonterminals	33	46	39.4%
	Productions	94	111	18.1%
	Total RHS	185	220	18.9%
Icon	Nonterminals	48	75	56.7%
	Productions	155	297	91.6%
	Total RHS	331	584	76.4%
Pascal	Nonterminals	37	46	24.3%
	Productions	84	93	10.7%
	Total RHS	187	184	-1.6%
Java	Nonterminals	155	196	26.5%
	Productions	312	587	88.1%
	Total RHS	561	1303	132.3%

Table 2: LALR(1) parsers generated from TCLLk-accepted grammars.

Grammar	shifts	reduces	states	shift/reduce conflicts	reduce/reduce conflicts
C	2836	260	342	2	2
EULER	1833	96	165	0	0
Icon	3429	162	270	7	0
Pascal	581	91	162	1	0
Java	4464	352	495	4	0

Figure 2. Estimated parse table sizes.

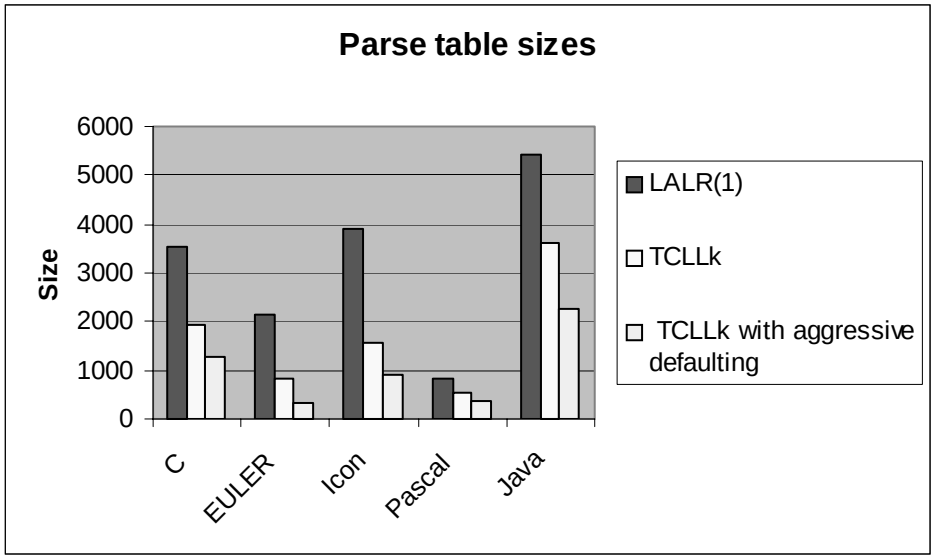

Table 3 Java grammar through TCLLk.

	Original LALR(1)	Input to TCLLk	After TCLLk
number of nonterminals:	160	155	196
number of productions:	331	312	587
number of symbols on right hand sides:	603	561	1303

.

Table 4 Backups in Java files.

Java file	backups	tokens in file	backups as % of tokens, initially	backups as % of tokens after creating ImportName
com.toolsofcomputing.SharedTableOfQueues	96	422	23%	0%
com.toolsofcomputing.FutureQueue	130	519	25%	<1%
java.util.Hashtable	366	1660	22%	1%
java.util.Vector	250	1268	20%	1%
java.util.StringTokenizer	97	454	21%	1%
java.util.BitSet	256	1329	19%	<1%
java.util.Date	472	2349	20%	3%
java.util.Calendar	346	2014	17%	2%