TCLLk vs. LR Parsing: A preliminary comparison

Thomas W. Christopher

Tools of Computing LLC
P.O. Box 6335
Evanston IL 60204-6335
http://www.toolsofcomputing.com

Tools of Computing LLC
Technical Report
1999-3-#3-TC
March 20, 1999
[revised March 31, 1999]

This document revised 3/31/99.

To contact the author

Thomas Christopher
Tools of Computing LLC
P.O. Box 6335
Evanston IL 60204-6335
USA

tc@toolsofcomputing.com
http://www.toolsofcomputing.com

To obtain a up-to-date copy of this document and the TCLLk parser generator

http://www.toolsofcomputing.com/freesoftware.htm

Acknowledgments

Some of the work on TCLLk was done at Illinois Institute of Technology, where the author is an Associate Professor.

Table of Contents

List of Figures

List of Tables

Summary

This paper is a preliminary comparison of the TCLLk parser generator and its algorithms to its main competitor, LALR(1). We seek the answer to this question:

> What is the likelihood TCLLk's parser generation algorithm will be preferable to LALR(1)?

We compare on four criteria:

1. How much work is required to get the grammar into a form the parser generator will accept?
2. What is the quality of parsers produced? How large are the parsers that it generates? How fast will they run?
3. How good is the parser's error recovery?
4. How much aid does the parser generator give in debugging grammars?

Our answers to these questions are:

1. TCLLk requires as little work to prepare grammars as LALR(1). Possibly less.
2. The quality, size and speed, of parsers produced is better than LALR(1). The size can be significantly less by using TCLLk's `-d` command line parameter.
3. The error recovery is easier and better than Yacc, and can be made much better.
4. Good diagnostics have not been developed yet for TCLLk.

We conclude it is likely that TCLLk's Strong LL(k) algorithms will prove preferable to LALR(1).

Foreword

This paper assumes you know something about algorithms for generating LL(1) and LALR(1) parsers. It also assumes you know something about TCLLk. TCLLk documents are available at

http://www.toolsofcomputing.com/freesoftware.htm

See particularly the documents:

- Thomas Christopher, *A Strong LL(k) Parser Generator That Accepts Non-LL Grammars and Generates LL(1) Tables*, Technical Report 1999-3-#2-TC, Tools of Computing, March 12, 1999.
- Thomas W. Christopher, *User Manual for TCLLk: A Strong LL(k) Parser Generator and Parser*, Technical Report 1999-3-#1-TC, Tools of Computing LLC, March 12, 1999

This presentation is incomplete. I should look up some references. Some information I am repeating from memory. (I'm sure readers will point out the bugs.)

I decided that it is more important to get this information out quickly than to perfect the presentation first. I'll be fixing the problems in later releases of this document.

Availability. TCLLk is available from Tools of Computing LLC at

http://www.toolsofcomputing.com/freesoftware.htm

The current version (20 Mar 1999) is an alpha version on the verge of becoming beta. It is written in the Icon programming language for use with a compiler witten in Icon. We (George Thiruvathukal and I) are preparing a postpass to generate parsers in Java, although we have no interest whatsoever to rewrite the parser generator itself.

-Thomas Christopher

Chapter 1 The competition between LL and LR

In this paper we are considering the TCLLk parser generator and the suitability of its algorithm for use in practical compilers. TCLLk is a Strong LL(k)[1] parser generator. When given an LL(1) grammar, it generates an LL(1) parser, so it subsumes LL(1).

Both LL and LR style parsing require deterministic context free grammars. They both parse in linear time.

The competition between them regarding which is the better algorithm is often expressed as "Which is more *powerful*?" Phrased properly, that question is susceptible to a mathematical treatment. It can be translated into "Which takes a larger class of languages?" or better, "Which can handle all the languages the other can and more?" This does not mean, "Which can handle the grammars the other one can?" You may have to find a different grammar for the same language. However, the real question defies a mathematical solution. The real question is, "Which is most convenient for the compiler writer?" Or, "What is the difference in the quality of parsers produced by these algorithms?"

Fischer and LeBlanc in their textbook, *Crafting a Compiler*[2], consider the question based on five criteria:

1. **Simplicity**. They point out that the underlying parsing algorithm may intrude on the user of a parser generator while the user is debugging his or her grammar. They conclude that LL(1) is better than LALR(1) by this criterion, since its underlying concepts are easier to understand. We will consider this in Chapter 5, Finding bugs in grammars, on page 36.
2. **Generality**. We are interested in
   - how many grammars can the algorithms can handle?
   - can one handle the grammars the other one can and more?"
   - is the algorithm powerful enough to handle all the interesting gram-

[1] The "Strong" in its name refers to its method for computing look-aheads of more than one symbol. It's not as "powerful," does not accept as many grammars, as a full LL(k) parser generator.

[2] Charles N. Fischer 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. They may have changed this section in more recent editions.

mars, i.e. programming language grammars?

They point out that while LL(1) and LALR(1) in practice are capable of handling most programming languages, programming language definitions are likely to come with LALR(1) grammars, not LL(1), and that it takes considerable effort to convert the grammars into LL(1) form.

They give LALR(1) the advantage over LL(1) by this criterion.

3. **Action symbols**. The question here is "How flexible is the interface to semantics routines?" LL(1) can call an action routine anywhere during the recognition of a right hand side of a production. LR algorithms can only call action routines while reducing a right hand side to a left hand side. However, LALR(1) can call actions in the same places by a simple transformation of the grammar, and many LALR(1) parser generators (e.g. YACC) will do the transformation themselves. LL(1) has only a slight edge here.

4. **Error repair**. Which parsers are better at recovering from or repairing errors in the input sentence? LL parsers have a stack of what symbols it expects to recognize later in the input, while LR parsers have a stack of states they were in earlier in the parse. The information LL parsers have is much easier to use to make good repairs. LL has a distinct advantage here.

5. **Table sizes**. How large are the parsing tables? Let |N| be the number nonterminal symbols, |T| be the number of terminal symbols, |P| be the number of productions, L be the sum of the lengths of the productions, and S be the number of states in an LR parser.

   - LL(1) parsers require an uncompressed table size of $|N|\times|T| + L$.

   - LALR(1) requires an uncompressed table size of $S\times(|N|+|T|)+2\times|P|$. In the worst case, LALR(1) can require as many as $O(2^L)$ states, making it totally unusable. They estimate, however, that in practical cases the number of states as about the same as the number of productions and the number of productions as about two per nonterminal.

   They estimate that the average case ratio of LALR(1) table size to LL(1) table size for typical programming languages, counting the difference in sizes of LL(1) and LALR(1) grammars for the same language and counting compression of the tables, is about 2 to 1. Overall, they give the advantage to LL(1).

In summary, in every way but in generality of grammars accepted, LL(1) is at least as good as LALR(1), or better.

Following Fischer and LeBlanc, we will discuss the convenience of TCLLk in comparison to LALR(1). We will try to answer these questions:

1. How much work is required to get the grammar into a form the parser

generator will accept?

2. What is the quality of parsers produced? How large are the parsers that it generates? How fast will they run?

3. How good is the parser's error recovery?

4. How much aid does the parser generator give in debugging grammars?

Some of these questions require qualitative answers and so are not easily formalized. We will consider each below.

Chapter 2 Grammars accepted

2.1 The LL/LR competition

No linear-time parsing algorithm will work with all context-free languages. In general it takes about cubic time to parse arbitrary context free languages. (Just a bit less in theory.)

So we can't expect an arbitrary language to be acceptable to either style parser generator. Worse yet, they place restrictions on the grammars they take. It can take a lot of work to manipulate a grammar into an acceptable form.

LL parser generators have traditionally been the losers on work required to manipulate its input grammar. Traditionally, the user has had to do left recursion removal and factoring. LR parser generators require neither.

TCLLk, however, does both left recursion removal and factoring itself, so that is no longer an advantage for LR parser generators.

What about generality of languages accepted? LR(1) parsers have been shown capable of parsing any deterministic context-free language. No LL(k) is, for any size k. Indeed, LL(1) is not as powerful as LL(2), nor in general is LL(k) as powerful as LL(k+1), and all of them are less powerful than LR(1).

Here is a killer grammar that is LR but not LL(k) for any k:

```
# Non-LL grammar
start = A | B.
A = x A y | x.
B = x B z | x.
```

The problem for LL parsers? It may see an arbitrary number of x's before discovering, by seeing a y or a z, whether its seeing an A or a B. LL has to know at the beginning.

Here's what TCLLk reports for it:

```
D:\Parsers\LLK>tcllk nonll
Warning: more than one empty RHSfound while factoring
"start:103"
Error: factoring "start:106" requires search of more
than 3 iterations
1 error and 1 warning
```


But the real question is more qualitative. “For programming language grammars, is there a significant advantage for either LL or LR parsers, or are they about equivalent?” The above grammar may not be typical of those we need to handle.

There is also another consideration: Full LR(1) parsers are not used because their table space is much too large. We use some more restricted version, SLR(1), NQLALR(1) a.k.a. SLALR(1), or LALR(1). What’s the difference? All of these use LR(0) parsing tables with some tricks for handling look-ahead. The parsing tables for LR parsers have one row for each state the parser can be in and a column for every terminal and nonterminal symbol. The number of states for LR(0) parsers is much smaller than for LR(1). Unfortunately, the fewer states make it impossible to keep as accurate look-ahead information as LR(1).

These are the tricks the practical LR algorithms use for look-aheads:

SLR(1) — “simple LR(1)” uses the symbols in the Follow sets to choose the action to take in “inadequate states.”

LALR(1) — “look-ahead LR(1)” uses all the information available in the graph of LR(0) states.

NQLALR(1) — “not quite look-ahead LR(1)” a.k.a. SLALR(1), “simple look-ahead LR(1),” are attempts to implement LALR(1). They contain a bug that everybody reinvents when they first attempt to implement LALR(1).

LALR(1) is better than either of the others, but NQLALR(1) is not always better than SLR(1) nor vice versa. LR(1) is more powerful than all of them, but of course, isn’t used because of its space requirements.

The idea that LALR(1) is better than LL(k) comes from three sources:

1. As mentioned, LALR(1) parser generators have in the past required less work from the user to get grammars into an acceptable form. This is no longer the case with TCLLk.

2. There is the example non-LL grammar for which LR is clearly superior. Indeed, that grammar is accepted by SLR(1) parsers generators. It may not be representative, however.

3. LR(1) is better than LL(k) for all k. However, we don’t use LR(1). We just assume that because it is more powerful, its simplifications will be as well. This is not warranted. There are grammars that TCLLk will accept that none of the simplified LR parsers will.

2.2 Testing TCLLk against LR grammars

There are a number of classic grammars that show the differences between various LR parser generators. We tried them on TCLLk to see about where TCLLk fits among the LR parsing algorithms. As it happened, TCLLk handled them all

with no problems. It didn't even have to resort to look-ahead of more than one symbol.

2.2.1 Non-SLR(1) but NQLALR(1)

Here is a grammar taken from the C programming language assignment statements:

```
#Lvalue.grm --
# taken from C assignment statements.
start = S EOF.
S = lvalue "=" expr.
S = expr.
lvalue = i.
lvalue = "*" expr.
expr = lvalue.
```

This grammar is not SLR(1). Upon encountering

```
* ii
```

Table 1 C assignments, NQLALR(1) but not SLR(1).

Item set 0: S' → .S EOF Successors: S → .lvalue = expr S ⇒ 1 S → .expr lvalue ⇒ 2 lvalue → .i expr ⇒ 3 lvalue → .* expr i ⇒ 4 expr → .lvalue * ⇒ 5 Item set 1: S' → S .EOF Successors: EOF ⇒ 6 Item set 2: S → lvalue .= expr Successors: expr → lvalue . = ⇒ 7 Item set 3: S → expr . Item set 4: lvalue → i .	Item set 5: lvalue → * .expr Successors: expr → .lvalue expr ⇒ 8 lvalue → .i lvalue ⇒ 9 lvalue → .* expr i ⇒ 4 * ⇒ 5 Item set 6: S' → S EOF . Item set 7: S → lvalue = .expr Successors: expr → .lvalue expr ⇒ 10 lvalue → .i lvalue ⇒ 9 lvalue → .* expr i ⇒ 4 * ⇒ 5 Item set 8: lvalue → * expr . Item set 9: expr → lvalue . Item set 10: S → lvalue = expr .

in the input, a SLR(1) parser first recognizes an lvalue, but it can't decide whether to convert it to an expr or to leave it an lvalue. The inadequate item set is

```
Item set 2:
S → lvalue .= expr
expr → lvalue .
Successor:  = ⇒ 7
```

If the parser sees "=" next, it should leave the lvalue as an lvalue, which is needed for the left side of an assignment expression, and read past the "=" going to state 7. If it sees an EOF, it should convert the lvalue to an expr.

Unfortunately, SLR(1) knows that both "=" and EOF can follow an expr, so it can't decide whether to reduce the lvalue to an expr when it sees an "=".

Both LALR(1) and NQLALR(1) can see from context that in that state, it should only reduce the lvalue to an expr if it sees an EOF.

What happens when we pass this grammar through TCLLk?

```
D:\Parsers\LLK>tcllk lvalue
0 errors and 0 warnings
```

We've added some action symbols to the grammar to see what TCLLk is doing. Here's the grammar:

```
#Lvalue.grm --
# taken from C assignment statements.
start = S EOF.
S = lvalue "=" expr P1!.
S = expr P2!.
lvalue = i P3!.
lvalue = "*" expr P4!.
expr = lvalue P5!.
```

Here's what TCLLk converts it into:

```
S = lvalue "S:101".
"S:101" = "=" expr P1.
"S:101" = P5 P2.
expr = lvalue P5.
lvalue = i P3.
lvalue = "*" expr P4.
start = S EOF.
```

2.2.2 LALR(1) but not NQLALR(1)

Here's a grammar from DeRemer and Pennello that is LALR(1) but not NQLALR(1).

```
# LALR(1)
start =   S .
```



```
S      =   a A c P1!.
S      =   a g d P2!.
S      =   b A d P3!.
S      =   b g c P4!.
A      =   B P5!.
B      =   g  P6!.
```

Its LR(0) item sets are shown in Table 2 Its inadequate sets are 6 and 9. In set 6, the parser should reduce the g to a B if it sees c next. Set 6 is only entered from set 2, where if g is reduced to B, the B will be reduced to an A, and the A will be followed by a c. Similar reasoning applies to set 9. The parser should reduce the g to a B if it sees d next. Set 9 is only entered from set 3, where after the reductions of g to B and B to A, the A will be followed by a d.

Table 2 LALR(1) but not NQLALR(1).

Item Set 0:⊥ S' → .S Successors: S → .a A c S ⇒ 1 S → .a g d a ⇒ 2 S → .b A d b ⇒ 3 S → .b g c Item Set 1: S' → S .⊥ Successors: ⊥ ⇒ 4 Item Set 2: S → a .A c Successors: S → a .g d A ⇒ 5 A → .B g ⇒ 6 B → .g B ⇒ 7 Item Set 3: S → b .A d Successors: S → b .g c A ⇒ 8 A → .B g ⇒ 9 B → .g B ⇒ 7 Item Set 4: S' → S ⊥ . Item Set 5: S→ a A .c Successors: c ⇒ 10	Item Set 6: S → a g .d Successors: B → g . d ⇒ 11 Item Set 7: A → B . Item Set 8: S → b A .d Successors: d ⇒ 12 Item Set 9: S → b g .c Successors: B → g . c ⇒ 13 Item Set 10: S → a A c . Item Set 11: S → a g d . Item Set 12: S → b A d . Item Set 13: S → b g c .

NQLALR(1) fails because, while searching backwards from set 6, reducing to B will take it to set 7. NQLALR(1) will then search back from set 7 to all its predecessors, seeing that the reduction of B to A will read a c next when it goes through set 2 and a d next when it goes through set 3. It decides that both c and

d can follow the B in set 6, so d won't choose whether to shift to set 11 or to reduce. (Similar reasoning applies to set 9.) NQLALR(1)'s failure comes from not remembering what set it arrived at set 7 from.

So anyway, what does TCLLk do with it? Here's the command line and response:

```
D:\Parsers\LLK>tcllk lalr1
0 errors and 0 warnings
```

Here's the translated grammar:

```
S = a "S:101".
S = b "S:102".
"S:101" = g "S:103".
"S:102" = g "S:104".
"S:103" = d P2.
"S:103" = P6 P5 c P1.
"S:104" = c P4.
"S:104" = P6 P5 d P3.
start = S.
```

2.2.3 LR(1) but not LALR(1)

Here's a grammar that is LR(1) but not LALR(1):

```
# LR(1)
S = a E c P1!.
S = a F d P2!.
S = b E d P3!.
S = b F c P4!.
E = f P5!.
F = f P6!.
start=S.
```

The item sets are shown in Table 3. I won't bother to explain this one. Here's what TCLLk does with it:

```
D:\Parsers\LLK>tcllk lr1
0 errors and 0 warnings
```

Yielding the translated grammar:

```
S = a "S:101".
S = b "S:102".
"S:101" = f "S:103".
"S:102" = f "S:104".
"S:103" = P5 c P1.
"S:103" = P6 d P2.
"S:104" = P5 d P3.
"S:104" = P6 c P4.
start = S.
```


Table 3 LR(0) item sets for the LR(1) grammar.

Item Set 0: S' → .S EOF S → .a E c S → .a F d S → .b E d S → .b F c Item Set 1: S' → S .EOF Item Set 2: S → a .E c S → a .F d E → .f F → .f Item Set 3: S → b .E d S → b .F c E → .f F → .f Item Set 4: S' ⇒ S EOF . Item Set 5: S → a E .c	Successors: S ⇒ 1 a ⇒ 2 b ⇒ 3 Successors: EOF ⇒ 4 Successors: E ⇒ 5 F ⇒ 6 f ⇒ 7 Successors: E ⇒ 8 F ⇒ 9 f ⇒ 7 Successors: c ⇒ 10
Item Set 6: S → a F .d Item Set 7: E → f. F → f. Item Set 8: S → b E .d Item Set 9: S → b F .c Item Set 10: S → a E c . Item Set 11: S → a F d . Item Set 12: S → b E d . Item Set 13: S → b F c .	Successors: d ⇒ 11 Successors: d ⇒ 12 Successors: c ⇒ 13

2.3 Java grammar

For a test of TCLLk against LALR(1) for a practical language, we cut and pasted an LALR(1) Java 1.1 grammar from The Java Language Specification[3] 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.

2.3.1 Putting the grammar into TCLLk syntax

We had to do a few cosmetic changes on their grammar. We passed the file through a small Icon program to change its syntax. By hand we changed their "one of" production form into a series of productions. We had the Icon program

[3] James Gosling, Bill Joy, Guy Steele, The Java Language Specification, available from www.javasoft.com.

create productions for all the symbols whose names end in “opt” indicating optional. The initial size of the Java grammar is shown in Table 4.

Table 4 Initial size of the Java grammar.

number of nonterminals:	160
number of productions:	331
number of symbols on right hand sides:	603

TCLLk reported errors:

```
9 errors and 0 warnings
```

2.3.2 Dangling else

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, so that’s what the Gosling *et al.* did.

They put in 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, as shown by these productions for if statements:

```
IfThenStatement =  if "(" Expression ")" Statement .
IfThenElseStatement =  if "(" Expression ")"
   StatementNoShortIf else Statement .
IfThenElseStatementNoShortIf =  if "(" Expression ")"
   StatementNoShortIf else StatementNoShortIf .
```

TCLLk can handle if-statements without this syntax, and it cannot handle them with it. We removed the “`NoShortIf`” versions of statements, a simplification of the grammar that removed nine productions.

2.3.3 Assignment expressions

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

We started with Primary and related nonterminals and 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. The left hand side is usually some low level of expression. In Java:

```
AssignmentExpression =  ConditionalExpression .
AssignmentExpression =  Assignment .
Assignment =  LeftHandSide AssignmentOperator
   AssignmentExpression .
```



```
LeftHandSide =  Name .
LeftHandSide =  FieldAccess .
LeftHandSide =  ArrayAccess .
```

A `ConditionalExpression` is a high level of expression, which can begin with the same kinds of things as `LeftHandSide`. Deep factoring is required to handle `AssignmentExpressions` — replacing nonterminals with their right hand sides repeatedly until 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 definition of `Assignment` with:

```
Assignment =  LeftHandSide AssignmentOperator
   AssignmentExpression .
LeftHandSide =  ConditionalExpression .
```

This will accept syntactically illegal constructs, but we can reject them in the semantics routines (easily, if we build a tree and generate code from it). This worked.

We then experimented with a lower level of expression:

```
LeftHandSide =  PostfixExpression.
```

which also worked. This led to a third version. We replaced the definition of `PostfixExpression`:

```
PostfixExpression =  Primary .
PostfixExpression =  Name .
PostfixExpression =  PostIncrementExpression .
PostfixExpression =  PostDecrementExpression .
PostIncrementExpression =  PostfixExpression
      "++" .
PostDecrementExpression =  PostfixExpression
     "--" .
```

with

```
PostfixExpression =  NameOrPrimary .
NameOrPrimary =  Primary .
NameOrPrimary =  Name .
PostfixExpression =  PostIncrementExpression .
PostfixExpression =  PostDecrementExpression .
PostIncrementExpression =  PostfixExpression
      "++" .
PostDecrementExpression =  PostfixExpression
     "--" .
```

and replaced the definition of `LeftHandSide`:

```
LeftHandSide =  NameOrPrimary.
```


This also worked, and this is the grammar we kept.

2.3.4 Switch bodies

The final problem was with switch statements. Their syntax is

```
SwitchStatement =  switch "(" Expression ")" SwitchBlock .
SwitchBlock =  "{" SwitchBlockStatementGroupsopt
   SwitchLabelsopt "}" .
SwitchBlockStatementGroups =  SwitchBlockStatementGroup .
SwitchBlockStatementGroups =  SwitchBlockStatementGroups
    SwitchBlockStatementGroup .
SwitchBlockStatementGroup =  SwitchLabels BlockStatements .
SwitchLabels =  SwitchLabel .
SwitchLabels =  SwitchLabels SwitchLabel .
SwitchLabel =  case ConstantExpression ":" .
SwitchLabel =  default ":" .
```

The problem was `SwitchBlockStatementGroupsopt` followed by `SwitchLabelsopt`. The `SwitchBlockStatementGroupsopt` can begin with `SwitchLabels` or can be empty. Similarly, `SwitchLabelsopt` can begin with `SwitchLabels` or can be empty. The two in a row confused TCLLk.

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

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

TCLLk's input translates this into the following productions:

```
SwitchBlock = "{" SwitchBlock_2_20.
SwitchBlockStatementGroup = SwitchLabels BlockStatements.
SwitchBlock_2_20 = SwitchLabelsopt "}".
SwitchBlock_2_20 = SwitchBlockStatementGroup
   SwitchBlock_2_20.
SwitchLabel = case ConstantExpression ":".
SwitchLabel = default ":".
SwitchLabels = SwitchLabel.
SwitchLabels = SwitchLabels SwitchLabel.
SwitchLabelsopt = SwitchLabels.
SwitchLabelsopt =.
SwitchStatement = switch "(" Expression ")" SwitchBlock.
```

The essential part of the translation is `SwitchBlock_2_20.` The simplest translation of a repetitive form, `{x}`, would yield a nonterminal, `A`, and two right hand sides. One right hand side would be `x A` and the other one would be empty.

Here the empty alternative has been replaced by the part of the `SwitchBlock` following it, i.e. `SwitchLabelsopt "}"`. This allows TCLLk to use factoring. TCLLk converted it to:

```
SwitchBlock = "{" SwitchBlock_2_20.
SwitchBlock_2_20 = SwitchLabels "SwitchBlock_2_20:102".
SwitchBlock_2_20 = "}".
"SwitchBlock_2_20:102" = "}".
"SwitchBlock_2_20:102" = BlockStatements SwitchBlock_2_20.
SwitchLabel = case ConstantExpression ":".
SwitchLabel = default ":".
SwitchLabels = SwitchLabel "SwitchLabels:101".
"SwitchLabels:101" = SwitchLabel "SwitchLabels:101".
"SwitchLabels:101" =.
SwitchStatement = switch "(" Expression ")" SwitchBlock.
```

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 5 shows the figures for the Java grammar before and after these by-hand transformations and after TCLLk got through with it.

Table 5 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

How long does TCLLk take to build and write out parse tables for Java? When executed in the command file:

```
writetime
tcllk Java
writetime
```

the difference between the two times written is usually 8 seconds (occasionally 7) 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.

2.4 Possible improvements in TCLLk

When fixing switch bodies in the Java grammar, we used the grammar input transformation to push a tail of an enclosing production into a subproduction, allowing factoring. This transformation can be used to handle nonterminals that have a conflict between the follow set for an empty production and the first set for some other right hand side. Currently this is always handled by building look-ahead trees that search beyond the end of the right hand side. With this transformation, these nonterminals can sometimes be handled by factoring.

With this transformation, we would not have had to rewrite the definition of switch statements.

2.5 Overall

TCLLk appears to be as convenient to prepare grammars for as LALR(1) parser generators. This conjecture is not susceptible to mathematical proof, nor perhaps experimentation, but it is possible to reach some consensus on with broader use of TCLLk or other parser generators using its algorithm.

Chapter 3 Size and speed of generated parsers

3.1 Fischer and LeBlanc's estimates

To recall Fischer and LeBlanc's discussion on the size of the parsing tables. Let |N| be the number nonterminal symbols, |T| be the number of terminal symbols, |P| be the number of productions, L be the sum of the production lengths, and S be the number of states in an LR parser.

- LL(1) parsers require an uncompressed table size of $|N|\times|T| + L$.
- LALR(1) requires an uncompressed table size of $S\times(|N|+|T|)+2\times|P|$. In the worst case, LALR(1) can require as many as $O(2^L)$ states, making it totally unusable. They estimate, however, that in practical cases the number of states as about the same as the number of productions and the number of productions as about two per nonterminal.

Of course, the size of an LL(1) grammar for a particular language is almost certain to be larger than the LALR(1) grammar one would use. Also, the parsing tables are typically compressed.

They include these rules of thumb for estimating the sizes of programming language grammars:

$|T| = |N|/2$

$|P| = 2\times|N|$

$L = 7\times|N|$

$S \approx |P|$

Only a fraction of the total number of possible table entries for either LL(1) or LALR(1) parsers are significant; the rest cannot be accessed while parsing a correct program, so they indicate an error has been detected. By only representing the nonerror entries, parsers can save a great deal of space. Fischer and LeBlanc estimate that 10% of LL(1) tables are non-error entries, and 5% of LALR(1) tables. Parser generators typically compress the tables. TCLLk does.

These approximations allow them to estimate parse table sizes as functions of the number of nonterminals:

Size of LL(1) tables $\approx 0.05\times|N|^2 + 7\times|N|$

Size of LALR(1) tables $\approx 0.15\times|N|^2 + 4\times|N|$

The limiting ratio of LALR(1) size to LL(1) size is 3 for grammars of the same size. That is not, however, the ratio for the same language, since we must also consider that LL(1) grammars are larger than LALR(1) for the same language.

They estimate that the average case ratio of LALR(1) table size to LL(1) table size for typical programming languages, counting the difference in sizes of LL(1) and LALR(1) grammars for the same language and counting compression of the tables, is about 2 to 1. For space required, they give the advantage to LL(1).

3.2 Sizes of parsers

3.2.1 Translated sizes of several grammars

TCLLk, by translating grammars to LL(1) form (perhaps with backups) allows us to start with one grammar, estimate the LALR(1) table size and compare to the LL(1) table size for the same language.

The size of a grammar increases when TCLLk (with some programmer intervention) translates a grammar from LALR(1) form to LL(k). How much of an increase can we expect? It is important; if we use Fischer's and LeBlanc's formulas for LALR(1) and LL(1) table sizes and if we can estimate the growth in grammar size, we can estimate whether TCLLk will produce smaller parsers than an LALR(1) parser generator, or larger, or about the same size.

Table 6 gives the increase in grammar size for a number of grammars when converted to LL(1) form by TCLLk. It should give some indication of what expansion can be expected. However, it is far from scientific. It doesn't provide a sample of programming language grammars, whatever that might be. Plus, the initial grammars were not perfectly LALR(1); Table 7 shows the number of conflicts Bison discovered.

Table 6 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%
EULER (hand translated by an expert)	Nonterminals	33	57	72.7%
	Productions	95	128	34.7%

Table 6 Increase in grammar size.

Grammar		Initial	After TCLLk	Growth
	Total RHS	262	326	24.4%
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%

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.

We passed the grammars through a translation program to put them in Yacc form and then passed them through Bison. Table 7 shows the number of shifts, reduces, states, and conflicts 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.

Table 7 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

3.2.2 Compressed tables

Recall that only a fraction of the total number of possible table entries for either LL(1) or LALR(1) parsers are significant; the rest, indicating input errors, can

be squeezed out to save space. Fischer and LeBlanc estimate that 10% of LL(1) tables are non-error entries, and 5% of LALR(1) tables.

In TCLLk, even greater compression can be specified. The TCLLk parser has a selection table in which it looks up nonterminals and terminals to decide which right hand side to replace the nonterminal with. It also has a default table. If it doesn't find the nonterminal/terminal pair in the selection table, it looks up the nonterminal in the default table to see if it is associated with a right hand side there. If there is only a single right hand side for a nonterminal, it will be in the default table, not in the selection table; no matter what the next symbol is, the nonterminal has to be replaced with that right hand side.

If the user specifies the -d flag to TCLLk, it will also use the default table for the right hand side associated with the greatest number of look-ahead terminals. This won't result in incorrect parsing; if the next terminal isn't correct, the parser will never recognize it. Errors will be detected at the same point in the program, but after several nonterminals have been replaced by their default right hand sides. Alone, this loses information for error recovery, but it does save space. TCLLk's parser, however, avoids the loss of information in a manner to be discussed below. Table 8 shows the selection table occupancy without and with the -d flag. It also shows the occupancy of the LALR(1) parsing tables for the grammars before they were transformed by TCLLk.

Table 8 Percent utilization of parse tables.

Language	**TCLLk without -d flag.[a]**	**TCLLk with -d flag.[b]**	**LALR(1) % occupied**
C	11.2%	4.9%	5.5
Euler	21.1%	2.6%	12.3
Icon	15.3%	4.3%	10.1
Pascal	8.4%	3.1%	5.1
Java	11.5%	4.1%	3.8

a.Percent of selection table utilized using default table only for single productions.
b.Percent of selection table utilized using default table for the most commonly selected RHS.

The Fischer/LeBlanc estimate of 10% occupancy of LL(1) tables and 5% occupancy of LALR(1) tables may be compared to these figures.

As mentioned, using the default table for the most common right hand side could result in a loss of information for error recovery: for every nonterminal expanded, i.e. replaced by a right hand side, the possibility of finding the other right hand sides is lost. TCLLk's parser, however, doesn't lose this information. It uses a technique taken from Burke-Fisher error recovery:

- When a nonterminal is expanded, the nonterminal is placed in a queue.

- When an action symbol is removed from the prediction stack, it is also queued.
- When a terminal is matched, the parser goes through the queue in FIFO order, removing nonterminals and performing the queued actions.
- An error is detected when either a terminal on the top of the prediction stack does not match the next symbol in the input or no right hand side can be chosen for the nonterminal on the top of the stack. Upon encountering the error, the parser restores the state it was in just after matching the previous terminal. It goes backwards (LIFO) through the queue removing the actions and nonterminals and doing the following:
  - it pushes each action symbol back on the prediction stack,
  - for each nonterminal, it removes the nonterminal's right hand side from the prediction stack and then pushes the nonterminal back on the prediction stack.

Since the parser is able to reconstruct all the information it had before expanding nonterminals, it has all the information it could have after matching the previous terminal. (Modifying the parser to also queue up a certain number of terminals would allow the parser to back up to the state it was in before matching the most recent one or more terminals. This is the first step towards implementing Burke-Fisher error recovery.)

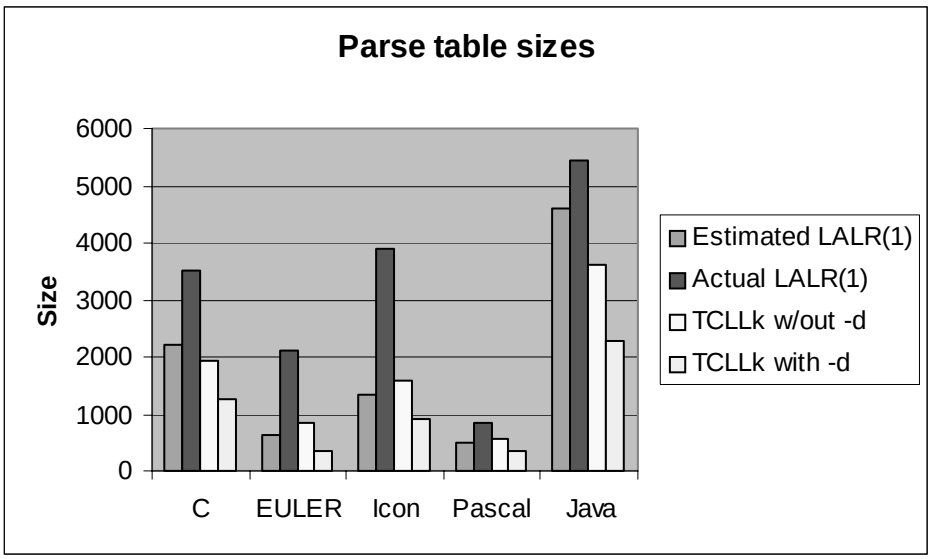
3.2.3 Comparing TCLLk tables to LALR(1)

Figure 1 shows a comparison of parse table sizes. The LALR(1) sizes are computed using Fischer's and LeBlanc's formula, both from their estimates and from the parsers produced by Bison. 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 tables. It omits the error recovery information. The units are not bytes. They are for crude comparison purposes only.

Figure 1 indicates several things:

- the actual LALR(1) parsers are larger than Fischer's and LeBlanc's estimates.
- TCLLk's parsers without the -d flag are comparable in size to Fischer's and LeBlanc's *estimates* for LALR(1) parsers, neither much better or worse.
- TCLLk's parsers without the -d flag are noticeable smaller than the actual LALR(1) parsers.
- TCLLk's parsers with the -d flag can be expected to be *much* smaller than LALR(1) parsers.

Figure 1 Parse table sizes (approximated).



3.3 Speed of parsers

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?

It 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 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 do 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-ups would be followed by reading 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 9 shows the number of tokens read and backups.

Table 9 Backups in Java files.

Java file	**backups**	**tokens in file**	**backups as % of tokens**
com.toolsofcomputing.SharedTableOfQueues	96	422	23%

Table 9 Backups in Java files.

Java file	backups	tokens in file	backups as % of tokens
com.toolsofcomputing.FutureQueue	130	519	25%
java.util.Hashtable	366	1660	22%
java.util.Vector	250	1268	20%
java.util.StringTokenizer	97	454	21%
java.util.BitSet	256	1329	19%
java.util.Date	472	2349	20%
java.util.Calendar	346	2014	17%

For a parse using the Java grammar, about one backup occurs for each five tokens read. For an examination of the translated grammar, the reason appears: a Name looks ahead beyond the Identifier. The cause appears to be

```
TypeImportOnDemandDeclaration = import Name "." "*"
   ";".
Name = SimpleName.
Name = QualifiedName.
QualifiedName = Name "." Identifier.
SimpleName = Identifier.
```

A "`. *`" can follow a Name, and a "`. Identifier`" can continue a Name. TCLLk has to look ahead to decide what to do in an import declaration.

An obvious optimization was to redefine ImportDeclarations as follows:

```
ImportDeclaration =  import ImportName ";" .
ImportName = Identifier "." ImportName.
ImportName = Identifier "." "*".
ImportName = Identifier .
```

After making these changes, the backups are as shown in Table 10. With a little

Table 10 Backups in Java files with nonterminal ImportName.

Java file	backups	tokens in file	backups as % of tokens
com.toolsofcomputing.SharedTableOfQueues	0	422	0%
com.toolsofcomputing.FutureQueue	2	519	<1%

Table 10 Backups in Java files with nonterminal ImportName.

Java file	backups	tokens in file	backups as % of tokens
java.util.Hashtable	11	1660	1%
java.util.Vector	11	1268	1%
java.util.StringTokenizer	5	454	1%
java.util.BitSet	6	1329	<1%
java.util.Date	74	2349	3%
java.util.Calendar	33	2014	2%

care the fraction of backups was made utterly trivial.

3.4 A grammar for exponential LALR parser size

Fischer and LeBlanc present the following grammar that can produce an exponential number of states ($O(2^n)$) in an LALR(1) parser with $O(n^2)$ productions:

```
start = S.
```
$S = X_i\ z_i$. *for* $1 \le i \le n$

$X_i = y_j\ X_i \mid y_j$. *for* $1 \le i,j \le n$ *and* $i \ne j$

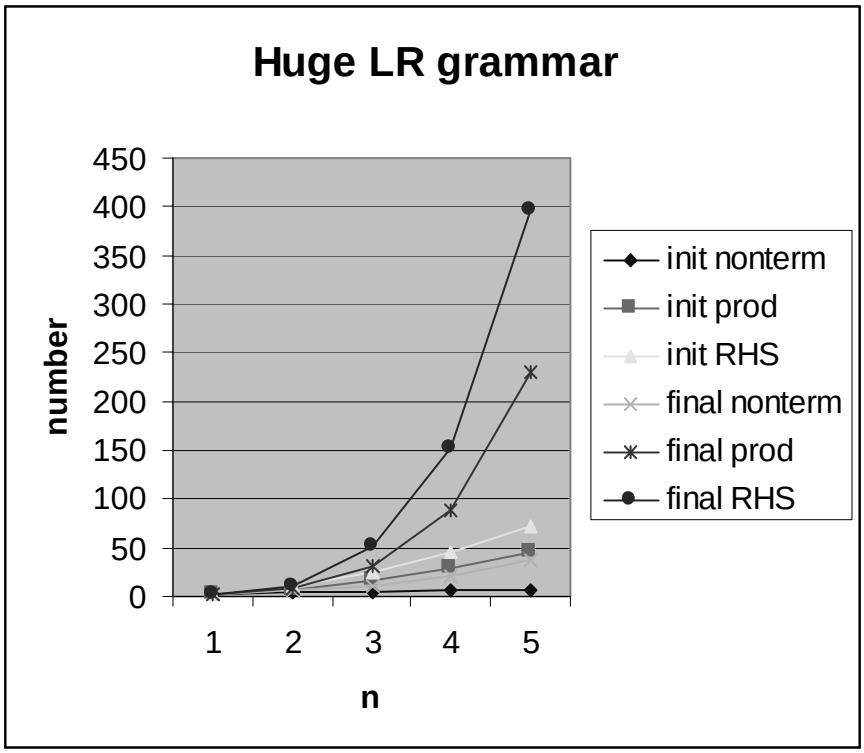
For example, for n=3, we have this grammar:

```
S = X1 z1.
S = X2 z2.
S = X3 z3.
X1 = y2 X1.
X1 = y2.
X1 = y3 X1.
X1 = y3.
X2 = y1 X2.
X2 = y1.
X2 = y3 X2.
X2 = y3.
X3 = y1 X3.
X3 = y1.
X3 = y2 X3.
X3 = y2.
start = S.
```

We shoved the grammar through TCLLk to see what would happen. For the n=3 grammar, we got:

```
S = y2 "S:104".
S = y1 "S:105".
S = y3 "S:106".
```


Figure 2 Sizes of TCLLk's translations of Huge LR grammar.




```
"S:104" = z1.
"S:104" = z3.
"S:104" = y3 "X1:101" z1.
"S:104" = y1 "X3:103" z3.
"S:104" = y2 "S:104".
"S:105" = z3.
"S:105" = z2.
"S:105" = y3 "X2:102" z2.
"S:105" = y2 "X3:103" z3.
"S:105" = y1 "S:105".
"S:106" = z1.
"S:106" = y2 "X1:101" z1.
"S:106" = z2.
"S:106" = y1 "X2:102" z2.
"S:106" = y3 "S:106".
X1 = y2 "X1:101".
X1 = y3 "X1:101".
"X1:101" = X1.
"X1:101" =.
X2 = y1 "X2:102".
X2 = y3 "X2:102".
"X2:102" = X2.
"X2:102" =.
X3 = y2 "X3:103".
```



```
X3 = y1 "X3:103".
"X3:103" = X3.
"X3:103" =.
start = S.
```

Figure 2 shows the growth of the resulting grammar sizes produced by TCLLk when given a series of these grammars.It is consistent with the hypothesis that TCLLk's algorithm also results in an exponential growth in grammar size, and hence in table size, in the worst case.

3.5 Possible improvements in TCLLk

Using Burke-Fisher[4] error recovery, parser operations (nonterminals replaced with right hand sides, terminal symbols recognized, and action symbols popped) are queued. When an error is detected, the queue allows the parser to back up to an earlier state while trying alternative repairs. This type of error recovery is not appropriate for interactive systems where actions must be performed immediately to respond to the user.

This queue allows the use of the default table without loss of error recovery capability. This results in a significant reduction in parse table sizes over LALR(1). It would also provide much better error recovery than either Yacc or TCLLk does currently.

Since TCLLk already uses a simpler version of the queue, it can do just as well with the -d flag as without it. Why do we need to specify the flag to get the smaller parsers? It is vestigial from TCLL1, the LL(1) parser generator and parser TCLLk was derived from, which did not use the queue. The -d flag will be eliminated in a future release.

3.6 Overall

With limited use of the default table, TCLLk appears comparable to LALR(1) with respect to parse table size. With extensive use of it, TCLLk can be significantly better. The use of Burke-Fisher error recovery can allow the parser to use the smaller size tables with an improved error recovery.

TCLLk is better with respect to parser size. It is probably not significantly worse with respect to parser speed.

---

[4] Burke, Michael, and Fisher, Gerald, "A practical method for syntactic error diagnosis and repair," *SIGPLAN Notices* 17(6). Also "A practical method for LR and LL syntactic error diagnosis and recovery," ACM *TOPLAS*, **9**(2) (1987), 164-197.

Chapter 4 Parser's error recovery

Error recovery is usually a lot better in LL than in LR parsers. LL parsers have a stack of the symbols they are expecting to match. LR parsers have their information hidden in the parse tables and a stack of the numbers of the states they were in. For a concrete comparison, we will examine TCLLk and Yacc.

4.1 TCLLk's error repair

TCLLk generates parsers with panic mode error repair.

The parser discovers an error in its input when the next input symbol either does not match the terminal symbol on top of the prediction stack or it does not select a right hand side for the nonterminal on top of the stack. There are no rules to tell the parser what to do next.

The parser gives an error message:

> unexpected token XXXX at line YYYY, column ZZZZ

The parser then attempts to resume parsing. There are two problems:

- The parser must get past the token that caused the syntactic error.
- The semantics routines must not become so confused that they either crash or flood the user with error messages. This requires that the semantics stack be set to an appropriate depth and that the contents of the stack not cause errors in the action routines. (Of course, the semantics could just be turned off.)

The simple error repair technique that TCLLk uses is *panic mode*. When the parser has detected and reported an error, it goes into panic mode and throws away part of the input until it has found a token in the input and a symbol in the prediction stack that allow parsing to continue. Using the prediction stack, it generates replacement text for the input that was thrown away. Then it returns to normal mode and continues parsing.

This leads to two questions:

- How does it choose an input symbol to restart at?
- How does it generate replacement text for the input it has thrown away?

The answers to the two questions are related.

The parser will read ahead to one of a set of symbols that delimit major sections of the program. These symbols are called *fiducial symbols*, symbols the parser can trust. For many programming languages, the fiducial symbols include ";", "then", "else", and "end", symbols that end or separate statements. If an error is detected within a statement, the parser will throw away the rest of the statement and try to resume parsing with the next.

The parser will not, however, accept just any fiducial. The fiducial must be predicted. The parser will throw away input symbols up to a fiducial and then look down the prediction stack. If it finds the fiducial symbol on the stack, or if it finds a nonterminal symbol that derives that fiducial symbol first in a string, then the parser will remove the symbols on the prediction stack down to the fiducial or nonterminal and will then resume parsing.

If the fiducial is not predicted, of course, the parser throws it away and continues looking. EOI (end of input) is a fiducial, and it is at the bottom of the stack, so the parser can at least resynchronize by throwing away all the rest of the program.

EOI is the only fiducial chosen by the parser generator. Users must specify the others with the fiducials declaration:

fiducial: f_1 f_2 ... f_n **.**

If the parser just throws away part of the prediction stack, the semantics stack will be mangled when the parsing resumes and the semantics routines will crash. Some parsers just turn off semantics on the first error. This is not a good solution for interactive systems.

The TCLLk parser tries to repair errors. After throwing away part of the input, it does *not* just throw away the top part of the prediction stack, but instead generates a replacement string of tokens for the input it has thrown away. Recall that an LL parser works by generating a program atop the input program, matching them. It is trivial to generate the replacement tokens. Instead of throwing away symbols from the prediction stack, it does the following with each top symbol of the prediction stack down to the symbol that predicted the fiducial:

- If the top symbol is a terminal, the parser generates an error token and pushes it onto the semantics stack. An error token can be recognized by the action routines. It warns the action routines that the token did not come from the user. The routines should not try to use the token nor give any further error messages.

- If the top symbol is an action symbol, the parser calls its action routine. The action routine will adjust the semantics stack properly. Most action routines will start by removing the correct number of values from the semantics stack and checking if there were any error tokens among them. If the action routine finds an error token, it will typically push the correct number of error tokens back on the stack (zero or one) and return immediately.

- If the top symbol is a nonterminal, the parser replaces it with one of its right

hand sides. The parser chooses the right hand side that will generate a shortest possible string of terminals. If there are several such right hand sides, the parser generator chooses arbitrarily which one will be used.

To summarize, TCLLk provides panic mode error *repair* with very little intervention from the user. The user only needs to specify some fiducial symbols.

4.2 Yacc's error recovery

What about LALR(1) parser generators? Here's what YACC does.

When *yyparse*, the parser generated by yacc, detects an error in the input, it calls the subroutine *yyerror()* (which the user provides) and then attempts to recover from the error.

Routine *yyerror* can be as simple as:

```
yyerror(msg) char *msg;
{printf(stderr,"%s\n",msg);}
```

The parser attempts to recover by the following method: It removes the top states from the state stack until it finds a state from which a shift of the token "error" is legal. It then pretends to find the "error" token and enters error mode.

It will remain in error mode until it has successfully recognized three consecutive tokens. While in error mode, if it detects another error, it will throw away the current token without generating an error message.

Panic mode error recovery can use such constructs as

```
statement:
          ...
          | error ';'
          ...
          ;
```

If the parser is within a statement when it discovers an error, it will come back to this state, pretend to read an error token, and look for a semicolon next. Since it is in error mode, it will keep reading tokens from the input and throwing them away until it finds a semicolon.

The user should never use "error" as a legitimate token in the program: it might confuse the error recovery algorithm. For a more elaborate error recovery, Figure 3 on page 35 shows some commands that can be put into yyerror.

Figure 3 Yacc commands to aid in error recovery.

yyerrok;	tells the parser to resume normal parsing mode (even before recognizing three successive tokens).
yyclearin;	tells the parser to clear its look-ahead token which it had read from the scanner. If you are reading ahead yourself (calling yylex), you must tell the parser to forget the last token it read and read a new one.
YYERROR	causes the parser to behave as if it detected an error.
YYACCEPT;	causes the parser to pretend it has accepted the entire input: yyparse returns 0.

To summarize, Yacc requires its user to modify the grammar and perhaps program some of the error recovery semantic actions using commands to control the parser itself. Yacc doesn't repair errors, but only recovers from them.

4.3 Possible improvements in TCLLk

Burke-Fisher[5] error recovery for non-interactive systems would improve TCLLk far beyond the state of the LALR(1) art. It also allows much smaller parsers than LALR(1).

4.4 Conclusion

For error recovery, TCLLk currently has the advantage over Yacc, being both easier to use and doing more. It also allows further improvements such as Burke-Fisher error recovery.

[5] Burke, Michael, and Fisher, Gerald, Op. cit.

Chapter 5 Finding bugs in grammars

5.1 Currently

Fischer and LeBlanc asserted that the LL(1) parsing algorithm is simpler than LALR(1)'s. Since this makes it easier to understand what is going on, LL(1) makes it easier to debug grammars, i.e. make them acceptable to the parser generator. This may be true for LL(1), but it is not true for TCLLk.

TCLLk's error diagnosis stinks.

TCLLk will report if any symbols cannot be derived from the start symbol. It will report if any nonterminals do not appear to generate finite strings of terminal symbols. It will report if the same nonterminal is both left and right recursive, making the grammar ambiguous. It will report if two or more right hand sides derive the empty string, another source of ambiguity.

Unfortunately, most of the errors reported are that

- factoring required more iterations than were permitted. Default is three iterations for the same nonterminal. It can be set to *n* by the command-line flag -f*n*.
- more than k symbol look-ahead was required. Default is 2. It can be set to *n* by the command-line flag -k*n*.

TCLLk is able to spew out volumes of information about intermediate grammars, but by the time the errors are reported, the grammar has been so rewritten that it is difficult to figure out what went wrong.

The best advice is to start with a subgrammar and add a few productions at a time. Errors then involve the new productions that were added. The TCLLk parser generator is so fast that there is no reason not to pass the grammar through it repeatedly.

The state of the art in reporting grammar errors in LALR(1) parser generators isn't that great, but it is better than TCLLk.

5.2 Possible improvements

TCLLk's diagnostics have not been the subject of much research yet, which can explain why they are poor. How might they be improved?

TCLLk can keep a data base on each transformation it performs, keeping track of why each nonterminal and production was created.

Just spewing out the history of the grammar productions that cause problems might swamp the user in too much information, but an interactive system could let the user browse and explore. Common patterns of grammar problems could be built in, allowing the system to make suggestions in many cases.

If nothing else, the experience of building and using an interactive parser development system would be instructive.

Chapter 6 Conclusions

We have compared TCLLk and LALR(1) parser generators with respect to four criteria:

1. **Generality**. How much work is required to get the grammar into a form the parser generator will accept?
2. **Parser size and speed**. What is the quality of parsers produced? How large are the parsers that it generates? How fast will they run?
3. **Parser error recovery**. How good is the parser's error recovery?
4. **Diagnosis of grammar problems**. How much aid does the parser generator give in debugging grammars?

To repeat what we concluded:

Generality. Neither TCLLk nor LALR(1) is a complete winner in generality of programming language grammars accepted, since each can handle grammars the other can not. However, several things suggest TCLLk's superiority:

- TCLLk's can handle grammars that cannot be handled by various LR algorithms.
- TCLLk can use greater than one symbol look-ahead,
- Converting the Java grammar from LALR(1) to TCLLk was painless.

TCLLk looks like it will require less effort to use than LALR(1) parser generators. This conjecture is unprovable mathematically and is not a good candidate for a controlled experiment, but it may be possible to reach a consensus on it.

Parser size and speed. TCLLk can be expected to produce much smaller parser tables than LALR(1). In speed, both are linear time parsers. TCLLk's look-ahead shouldn't intrude much.

Parser error recovery. TCLLk's parser error recovery is superior to LALR(1). Adding Burke-Fisher error recovery can make it spectacularly better then LALR(1).

Diagnosis of grammar problems. TCLLk currently has very poor diagnostics. This area hasn't been researched yet, so there is the possibility for dramatic improvements.

Overall. TCLLk's parser generation algorithm is already competitive with LALR(1) and is likely to become much better. We speculate that TCLLk will eventually replace LALR(1) in actual use.

