User Manual for TCLLk: A Strong LL(k) Parser Generator and Parser

Thomas W. Christopher

Tools of Computing LLC

Tools of Computing LLC
Technical Report
1999-3-#1-TC
March 12, 1999

This document revised 12/31/99.

To contact the author:

Thomas Christopher
School of Computer Science, Telecommunications and Information Systems
DePaul University
243 South Wabash Ave.
Chicago IL 60604-2301 USA

tchristopher@cs.depaul.edu
http://bach.cs.depaul.edu/tchristopher

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

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

Acknowledgments

I wish to thank Patricia Guilbeault for her technical editing of this document.

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

Chapter 1 Introduction

Parsing, finding the phrases in a program, is the first job of a compiler. The LL(1) parsing algorithm is probably the easiest parsing algorithm to understand and the easiest program for error recovery. The "(1)" in LL(1) indicates that the parser gets to look ahead one symbol during the parse. But LL(1) parsing algorithms require skill to use. They require that the programming language grammar be rewritten, usually extensively, to be put in LL(1) form.

LL(k) parsing, where the parser gets to look k symbols ahead, is the generalization of LL(1). The larger the value of k, the more "powerful" the parser is, that is to say, the more grammars it can handle. Unfortunately, LL parsers are table driven, and the usual design of LL(k) parsers requires much too much table space for any k>1. If the number of terminal symbols is t, they require $O(t^k)$.

This document describes the use of TCLLk, a Strong LL(k) parser generator and parser, written in the Icon programming language. The parser generator builds tables for the parser to use. Its most important features are:

(1) It rewrites the grammar itself to try to put it into LL(1) form. This makes it a lot easier to use than most LL(k) parser generators.

(2) It resorts to look-ahead of greater than one symbol only when it has to.

(3) It converts the extra look-ahead into look-ahead trees using LL(1) grammar productions. The TCLLk parser uses LL(1) tables. The size of the tables is, in practice, much smaller than they would be with a traditional LL(k) parser generator.

(4) Like TCLL1 (an earlier LL(1) parser generator), TCLLk uses "action symbols" to interface to the semantics routines of a compiler.

(5) Like TCLL1, TCLLk provides "panic mode" error repair.

Topics in this document include

1. how to build the parser generator,
2. how to write the grammar
3. how TCLLk rewrites it into LL(1) form,
4. how the parser handles error recovery, and

5. how to interface to the parser to the compiler's semantics routines.

The goal of this document is to give compiler writers the training they need to use the TCLLk parser generator to build parsers for their compilers. In an attempt to be self contained, it includes a brief introduction to context free grammars, so many readers will wish to skim or skip parts of this document.

TCLLk is an improvement on TCLL1. Some of the discussion will show what TCLL1 does with a grammar to contrast it to what TCLLk does. Some of the program files and data structures are the same for both parser generators. This document itself is a modification of the document TCLL1: An LL(1) Parser Generator and Parser, by the same author.[1]

[1.] It may take a few passes to clean up this document, updating the parts of the discussion from TCLL1 to TCLLk.

Chapter 2 Building the parser generator

Before reading the rest of this description of TCLLk, you should compile it on your own system. That will allow you to try out the test grammars as they are discussed.

If you do not have a copy of Icon, you can get it over the Internet through the World Wide Web at, *http://cs.arizona.edu/icon/*.You may also want to pick up the Icon Programming Library.

If you have the Icon Programming Library (IPL) installed on a WINDOWS machine, you can execute the batch file buildk.bat (or buildknt.bat) to build the parser generator. The four files from the IPL that the parser generator uses are included with this distribution and can be compiled separately. To build the parser generator by hand, you may execute

```
rem These are from the Icon Program Library:

icont -c xcode escape ebcdic options pathfind

rem These form the parser generator proper

icont -c grananal llk semstk readllk parsellk
icont -c scangram semgramk
icont -fs tcllk
```

The first *icont* line compiles the files from the IPL. You may omit the line if you have the IPL installed. The second *icont* line compiles modules used by the parser generator. The third line compiles the parser generator's main program. The flag *-fs* tells the translator that the parser generator calls some procedures by giving their names as strings.

To use TCLLk to build a parsing table, execute

tcllk *grammar.grm*

where *grammar.grm* is the grammar file. The output of the parser generator will be encoded parse tables in file *grammar.llk*.

You can also specify flags and options on the command line

tcllk *options grammar.grm*

Options in TCLLk's command line are

-v for verbose output, giving the grammar after every transformation

-p for a listing of the grammar productions before and after transformation

-e for a grammar listing after all transformations have been done, including information on sets of symbols used to build the parse tables

-s to write out “statistics” after the various transformations.

-d to choose some productions by default if no others are selected (this produces smaller table size, but not as good error recovery).

-k*n* for integer *n* restricts the look-ahead to no more than *n* symbols. The default is 2.

-f*n* for integer *n* restricts the number of “deep factoring” iterations to no more than *n* for a particular nonterminal. The default is 3.

TCLLk reads its own parsing table from file *tcllk.llk* which must be in the current directory or in the path bound to the environment variable LLPATH, or if that isn’t set, then environment variable DPATH. The directory names in the path are **separated by blanks**, an Icon Program Library convention.

Chapter 3 Context-Free Grammars

3.1 Context-free grammars

To use LL(1) parsers, you need to be skilled at manipulating context-free grammars. TCLL1 is typical of LL(1) parsers. TCLLk, however, does a lot of the work for you. Alas, the difficult rewritings are difficult in both systems.

In any event, you must write context free grammars (CFGs) to give to the parser generators. Here is a discussion of CFGs.

In the jargon of formal language theory, a context-free grammar (CFG) is a "4-tuple" (N,T,s,P). That is to say, it has four parts:.

- T—a set of terminal symbols, the words in the language;
- N—a set of nonterminal symbols which do not themselves appear in sentences, but are used to generate sentences.
- s—one of the nonterminals, the start symbol.
- P—a set of productions, the rules for generating sentences.

(In a later section, we will add another part: a set of action symbols.)

A production of a CFG is typically written

$$LHS \rightarrow RHS$$

Where LHS, the "left hand side" of the production, is a single nonterminal symbol and RHS, the "right hand side" of the production is a string of zero or more symbols, terminal and nonterminal.

The arrow is a metalinguistic symbol: it is used to write the productions; it is not part of the language they describe.

Since we are using a particular parser generator, TCLLk, we will follow its input syntax and write the productions as

LHS = *RHS* **.**

with the equal sign and period as metalinguistic symbols.

We will speak of a left hand side as “possessing” the productions or the right hand sides of the productions it appears in. We will also speak of the right hand side being a right hand side “for” the left hand side symbol.

The productions are rewriting rules. A sentence is generated by starting with a string composed solely of the start symbol and repeatedly replacing a nonterminal in the string with one of its right hand sides. When there are only terminal symbols left in the string, the string is called a *sentence* in the language.

Each single rewriting is called a *derivation step*. A sequence of derivation steps, especially the sequence leading from the start symbol to a sentence is a *derivation*. A derivation step is written

$$uAw \Rightarrow uvw$$

where $A = v$. is a production and u and v are any strings of zero or more symbols. A derivation composed of zero or more derivation steps is written:

$$u \Rightarrow^{*} w$$

In TCLLk, a nonterminal or terminal symbol is written as an identifier or as any string of printable symbols surrounded by quotes. An identifier is the same symbol whether it is quoted or not, i.e. X is equivalent to “X”.

We will use the expression grammar Figure 1 in many of our examples.

Figure 1 *Expression grammar*

```
start = e .
e = e "+" t .
e = e "-" t .
e = t .
t = f "*" t .
t = f "/" t .
t = f .
f = i .
f = "(" e ")" .
```

TCLLk assumes the start symbol is named start. Since we really wanted e to be the start symbol, we put in a production

```
start = e .
```

The terminal symbols are *“(", ")", "*", "+", "-", "/"*, and *i*.

The nonterminal symbols are *start*, *e*, *t*, and *f*.

3.2 Derivations

All strings derived from the start symbol are called *sentential forms*. A *sentence* is a sentential form composed entirely of terminals.

Figure 2 an example of the derivation of a sentence.

Figure 2 The derivation of a sentence

start
e
t
f*t
(e)*t
(e-t)*t
(t-t)*t
(f-t)*t
(i-t)*t
(i-f*t)*t
(i-i*t)*t
(i-i*f)*t
(i-i*i)*t
(i-i*i)*f
(i-i*i)*i

Not only was this a derivation, it was a *leftmost derivation*; we always replaced the leftmost nonterminal in the string with a right hand side. We could have done a *rightmost derivation*, always replacing the rightmost nonterminal. Or we could have replaced arbitrary nonterminals. Since the nonterminals are replaced without regard to the symbols that surround them, the sentences we can derive don't depend on the order of replacement. That is the meaning of "context-free". However, the set of sentential forms we can derive do depend on the order of replacement.

A leftmost derivation step is written

$$uAw \Rightarrow_L uvw$$

where *u* must be composed entirely of terminals and $A = v$. is a production. Similarly, $\Rightarrow_L{}^*$ represents a leftmost derivation; $\Rightarrow_R$, a rightmost derivation step; and $\Rightarrow_R{}^*$, a rightmost derivation.

An LL parser finds a leftmost derivation of the input sentence.

3.3 Phrases

A *phrase* is a substring of a sentential form that was derived from a single nonterminal during the derivation of the sentential form. When we compile a pro-

gram, we will deduce the meanings of phrases from the meanings of the words and phrases contained within them. In the sentence *(i-i*i)*i* derived above, the substring *i-i*i* is a phrase; it was derived from an *e*. Within the *i-i*i*, *i*i* is a phrase, but *i-i* is not. Although the string *i-i* could be derived from an *e*, in this derivation it was not.

Notice that for two phrases in a sentential form, one of the following is true:

- one is a substring of the other, or
- they have no symbols in common, or
- they are the same string.

3.4 Bugs in grammars

Context-free grammars can have bugs in them. The grammar in Figure 3 exhibits three common bugs.

Figure 3 Grammar bugs.grm

```
start = e.
e = e "+" t .
e = e "-" t .
t = t "*" t .
t = t "/" t .
t = f .
p = i .
p = "(" e ")" .
```

If we pass this grammar through TCLLk, we get the error messages shown in Figure 4.

Figure 4 Error messages from bugs.grm (Figure 3).

```
Error: start does not appear to derive a terminal string
Error: e does not appear to derive a terminal string
Warning: p cannot appear in a sentential form; it will be remnoved.
Warning: ) cannot appear in a sentential form; it will be remnoved.
Warning: ( cannot appear in a sentential form; it will be remnoved.
Warning: i cannot appear in a sentential form; it will be remnoved.
2 errors and 4 warnings
```

We got the first two errors by removing the production "*e = t.*". Once we have the symbol *e* in a string, we can't get rid of it: *e* can only be replaced with strings containing an *e*. Since *start* only goes to *e*, *start* too cannot derive a string composed entirely of terminals.

The reason TCLLk says the symbols do not *appear* to derive a terminal string has to do with its algorithm. It tries to calculate the minimum length of a string

of terminals the nonterminal can derive. If it can't satisfy itself that the nonterminal can generate a string of less than a certain large length, it reports the error. You can fool it by writing a grammar that will only generate sentences of greater than that length.

The errors stating that *p*, *")"*, *"("*, and *i* cannot appear in a sentential form simply means that there is no sequence of derivation steps starting from the start symbol that can derive a string containing those symbols. The bug may be that we should have used *f* rather than *p* in the last two productions, or we should have had some more productions for *f* including "*f = p*.".

3.5 Ambiguous grammars

So lets fix the errors in bugs.grm (see Figure 3 on page 16), getting the grammar LnRRecBug.grm (Figure 5).

Figure 5 Grammar LnRRecBug.grm

```
start = e.
e = e "+" t .
e = e "-" t .
e = t.
t = t "*" t .
t = t "/" t .
t = p .
p = i .
p = "(" e ")" .
```

TCLLk gives the error messages shown in Figure 6.

Figure 6 Error message from LnRRecBug.grm (Figure 5).

```
Error: t is both left and right recursive, the grammar is ambiguous
1 error and 0 warnings
```

The error

```
Error: t is both left and right recursive, the grammar
is ambiguous
```

reports on one of the most common bugs in grammars used for programming languages. To understand it, we need a few concepts first.

A sentence derived using a particular grammar is *ambiguous* if there is more than one way to divide it up into phrases. It can be proven that the sentence is ambiguous if and only if it has more than one leftmost derivation. A grammar is ambiguous if it can generate any ambiguous sentences.

Since the phrases of a sentence are used to determine its meaning, an ambiguous sentence can have more than one meaning. An ambiguous programming language grammar would militate against reliable software.

An additional problem for compiler writers is that there are no fast parsing algorithms that work for ambiguous grammars. (Some parser generators will accept ambiguous grammars, but most of them resolve the ambiguity internally before generating the parsers.)

It would be nice if we could find out if a given grammar is ambiguous. Unfortunately, it is impossible to do that in general. It is *incomputable* whether an arbitrary context free grammar is ambiguous. There is no algorithm that can take an arbitrary context free grammar and report whether it is ambiguous or not. It is not, mind you, that we don't know the algorithm yet. We can prove that there is no such algorithm possible.

For particular grammars we may be able to prove they are ambiguous or unambiguous, but there will always be some that we cannot be sure about. Here are two classes of grammars we *can* know about:

- If the grammar is accepted by TCLLk without any warnings or errors, it is unambiguous.
- If the grammar is left and right recursive in the same nonterminal, it is ambiguous.

A nonterminal is left recursive if it can derive a string in which it appears as the leftmost symbol. It is right recursive if it can derive a string in which it appears as the rightmost symbol. Don't consider only leftmost derivations for this definition of right recursive: the symbol might be followed by some nonterminals that derive the empty string that we have to get rid of. Consider the grammar

```
A = i A B | i.
B = .
```

A is right recursive which can be seen from the rightmost derivation

```
A
i A B
i A
i i
```

but not from the leftmost derivation

```
A
i A B
i i B
i i
```

If a grammar is both left and right recursive in the same nonterminal then the grammar is ambiguous. As a proof, suppose A is both left and right recursive in

a reduced grammar (i.e., a grammar without bugs in it), then there are strings v, w, x, y, and z such that:

$$A \Rightarrow_L^* A\ v$$
$$A \Rightarrow_L^* x\ A\ w$$
$$A \Rightarrow_L^* y$$
$$w \Rightarrow_L^*$$
$$v \Rightarrow_L^* z$$

where

$$v \in (N \cup T)^*$$
$$x \in T^*$$
$$w \in N^*$$
$$y \in T^*$$
$$z \in T^*$$

(that is to say, x, y, and z are strings of terminals, w is a string of nonterminals that can derive the empty string, and v is a string of any symbols).

This allows two different leftmost derivations of *xyz* from *A* shown by these sentential forms along the derivation:

A	A
$\Rightarrow_L^*$ A v	$\Rightarrow_L^*$ x A w
$\Rightarrow_L^*$ x A w v	$\Rightarrow_L^*$ x A v w
$\Rightarrow_L^*$ x y w v	$\Rightarrow_L^*$ x y v w
$\Rightarrow_L^*$ x y v	$\Rightarrow_L^*$ x y z w
$\Rightarrow_L^*$ x y z	$\Rightarrow_L^*$ x y z

Since a nonterminal being both left- and right-recursive is a common bug in programming language grammars, TCLLk checks for it, but remember, there are many other ways for grammars to be ambiguous as well.

3.6 Leftmost derivation algorithm

Sentences can be generated with a leftmost derivation using a *prediction stack*. The algorithm is shown in Figure 7 on page 20.

We call the stack the prediction stack since it predicts what symbols and phrases will be generated later.

Figure 7 Leftmost derivation algorithm.

THE LEFTMOST-DERIVATION ALGORITHM

Initially, place the start symbol on the prediction stack.

Repeat

pop the top symbol off the prediction stack

if it is a terminal, write it out

if it is a nonterminal, then choose one of its right hand sides and push it on the prediction stack, leftmost symbol on top

until the prediction stack is empty.

3.7 Extended syntax

To make it more convenient to write grammars, TCLLk provides an extended syntax for expressing alternatives, groupings, optional parts, and repetitions. We show how they may be used with the expression grammar given above:

| the vertical bar is used to separate alternatives. It is customarily used to combine all the productions for a single nonterminal into a single rule. It can also separate alternatives within groupings. We can, for example, shorten our expression grammar from nine lines to four:

```
start = e .
e = e "+" t  | e "-" t | t .
t = f "*" t | f "/" t | f .
f = i | "(" e ")" .
```

( ) parentheses are used to group symbols and alternatives. We can group the operators in our expression grammar as follows:

```
start = e .
e = e ("+" | "-") t | t .
t = f  ("*" | "/") t | f .
f = i | "(" e ")" .
```

[ ] brackets group optional items; [x] is equivalent to (x |); that is, a bracketed item is equivalent to the enclosed item or the empty string. In our expression grammar, the alternatives for *t* provide an optional part:

```
start = e .
e = e ("+" | "-") t | t .
t = f  [ ("*" | "/") t ].
f = i | "(" e ")" .
```


{ } braces group items that may occur any number of times. The alternatives for *e* provide an example of this repetition:

```
start = e .
e = t { ("+" | "-") t } .
t = f  [ ("*" | "/") t ].
f = i | "(" e ")" .
```

When given a grammar using the syntax extensions, TCLLk translates it into a pure, unextended CFG. It does this by introducing new nonterminals for all the groupings. It constructs the names of the new nonterminals from the left hand side symbol, line number, and position on the line where the grouping begins:

```
LHS_lineNumber_column
```

You should be able to figure out how grammars are transformed from the extended notation to the basic notation by comparing our final expression grammar

```
start = e .
e = t { ("+" | "-") t } .
t = f  [ ("*" | "/") t ].
f = i | "(" e ")" .
```

to its transformed version:

```
e = t e_2_7.
e_2_7 = e_2_9 t e_2_7.
e_2_7 =.
e_2_9 = "+".
e_2_9 = "-".
f = i.
f = "(" e ")".
start = e.
t = f t_3_8.
t_3_10 = "*".
t_3_10 = "/".
t_3_8 = t_3_10 t.
t_3_8 =.
```

TCLLk provides one further enhancement to CFGs: *action symbols.* Action symbols provide the interface between the parser and the "semantics" in the compiler. An action symbol is written as an identifier followed by an exclamation point:

```
ID !
```

As far as the language generated from the grammar is concerned, action symbols don’t appear. They behave as if they were nonterminals that only have one production and that production has an empty right hand side. During the parse, however, whenever the parser finds an action symbol on the top of the predic-

tion stack, it performs some action as it pops the symbol off. In a later section, we will discuss the use of action symbols.

Chapter 4 LL Parsing

4.1 Principles of LL(k) Parsing

"LL(k)" means the parser works **L**eft-to-right, finding a **L**eftmost derivation of the sentence, and looking at most **k** symbols ahead in the input to decide what action to take next.

The trick is this: the LL(k) parser generates a sentence on top of the input sentence, matching the two. When it has successfully matched all of the input sentence, it has also parsed it, since the phrases of the input are the same as those of the generated sentence.

If we are only interested in whether the input is a sentence—and not interested in the phrases—we call the parser a *recognizer*. We present an LL(1) recognizer now, and wait to present TCLL1 and TCLLk parsers until we have discussed action symbols. The sentence generation algorithm of the last section now becomes the recognition algorithm shown in Figure 8.

Figure 8 LL(1) Recognition algorithm.

THE LL(1) RECOGNITION ALGORITHM

Initially, place the start symbol and the EOI (end of input) symbol on the prediction stack with the start symbol on top. Append EOI to the right end of the input. Read the leftmost symbol from the input into the *current symbol*.

Repeat

pop the *top symbol* off the prediction stack

if it is a terminal, compare it to the *current symbol*. If they match, read the next input symbol into the *current symbol*. If they don't, an error has been discovered in the input.

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

until the EOI symbol is matched.

The EOI ("end of input") symbol is inserted into the set of terminals by the TCLLk parser generator. It is used to permit the parser to recognize when the last terminal has been read in. In a real compiler, the parser calls the scanner to return one symbol of the sentence at a time, left to right. The scanner will return EOI when there are no more symbols in the input.

The parser must decide what string of symbols to replace a nonterminal symbol with by looking at the symbol and the next terminal symbol in the input. The easiest data structure to use is a two dimemsional array indexed by the nonterminal and the terminal, requiring $|N|\times|T|$ space.

In LL(k) parsers, the LL(1) recognition algorithm is modified to look at the nonterminal from the top of the stack and the next k symbols in the input. Traditionally, LL(k) parsing uses an array of k terminal symbols from the input and looks up the replacement for a nonterminal in a table of size $|N|\times|T|^k$, indexing it the nonterminal and the k symbols in the array. The k-symbol look-aheads are really kept as arrays of k symbols. TCLLk, however, builds look-ahead trees only where the look-aheads are actually needed. An LL(k) recognition algorithm is shown in Figure 9.

Figure 9 LL(k) Recognition algorithm.

THE LL(k) RECOGNITION ALGORITHM

Initially, place the start symbol and k copies of the EOI (end of input) symbol on the prediction stack with the start symbol on top. Append k copies of the EOI to the right end of the input. Read the leftmost k symbols from the input into the *current symbol* array. The *current symbol* array is k elements long, indexed from 1 to k.

Repeat

pop the *top symbol* off the prediction stack

if it is a terminal, compare it to the *current symbol[1]*. If they match, remove *current symbol[1]*, shift the other elements of *current symbol* down one position, and read the next input symbol into the *current symbol[k]*. If they don't match, an error has been discovered in the input.

if it is a nonterminal, then choose one of its right hand sides and push it on the prediction stack, leftmost symbol on top. Choose the right hand side by looking at the *current symbol* array and deciding which RHS will allow parsing to continue. This requires a table of size $|N|\times|T|^k$.

until the EOI symbol is matched.

4.2 What TCLLk does

The big problem in LL(1) parsing is that the grammar has to be rewritten into *LL(1) form*, a form in which the next input symbol can always tell the parser

which right hand side to choose. In subsequent sections we will discuss ways to rewrite grammars to put them in LL(1) form.

TCLLk works in two steps:

1 It rewrites the grammar to try to put it in LL(1) form.

  This saves you from having to rewrite the grammar yourself. Nevertheless, you may need to understand how it is done by hand to understand the rewritten grammar that TCLLk shows when something is still wrong. There are some rewrites that we will show that TCLLk does not perform. You may need to use these yourself if TCLLk's transformations don't bring the grammar all the way to LL(1) form.

2 If step (1) doesn't bring the grammar all the way to LL(1) form, TCLLk will build look-ahead trees of up to k symbols in depth. The trees are built using LL(1) productions, so the parser can use LL(1) parsing tables.

4.3 Putting grammars into LL(1) form

It is hard to put grammars into LL(1) form. Here we consider the requirements LL(1) places on grammars, how grammars fail to meet those requirements, and techniques for rewriting grammars to make them suitable.

First a caution. To save yourself much grief, obey this simple rule when transforming grammars yourself: You may introduce new nonterminals. You may revise the definitions of existing nonterminals. You may delete nonterminals if they are no longer needed. But *never* change the meaning of a nonterminal. Never change the set of strings a nonterminal generates.

4.3.1 How a grammar fails to be LL(1)

The only place in the LL(1) recognition algorithm where problems can arise is when a nonterminal comes to the top of the prediction stack. The parser must pick one of the nonterminal's right hand sides to replace it with, a right hand side that will allow parsing to continue. To do this, it can look only at the next symbol in the input. As an example of where this fails, consider our expression grammar productions for *t* and *f*:

```
t = f "*" t .
t = f "/" t .
t = f .
f = i .
f = "(" e ")" .
```

Suppose *t* is on top of the prediction stack. Each of its right hand sides begins with *f*. An *f* itself can begin with either an *"("* or an *i*. So if we see either an *"("* or an *i* next in the input, we can't possibly tell which production for *t* we should use.

We need two concepts:

- The *first set* of a string of symbols, *u*, is the set of terminal symbols, *First(u)*, that can occur leftmost in a string derived from *u*. In formal notation

  First(u) = { a | u$\Rightarrow^{*}$ av , a *is a terminal symbol*, u *and* v *are strings*}

- The follow set of a nonterminal, *A*, is the set of symbols, *Follow(A)*, that can follow *A* in a sentential form. Formally

  Follow(A) = { b | s $\Rightarrow^{*}$ v A b w , s *is the start symbol*, b *is a terminal symbol*, v *and* w *are strings*}

Now lets consider which right hand side to choose for a nonterminal. Given a production

A = u .

what terminal symbol, *t*, would tell us to replace *A* with *u*?

- If symbol *t* is in *First(u)*, we should choose *u*. After all, we want to choose the right hand side that will allow us to continue parsing, and right hand side *u* will at least be able to get past the next input symbol.

- If *u* is the empty string, or if it derives the empty string, and if *t* is in *Follow(A)* we should choose *u*. After all, if the right hand side vanishes, the next input symbol we are looking at could be one that follows the phrase, not one that begins it.

If any terminal symbol tells us to choose more than one right hand side for a nonterminal, the grammar is *not* LL(1). If no terminal symbol ever tells us to choose more than one right hand side for any nonterminal, the grammar *is* LL(1).

When we give a grammar that is not LL(1) to TCLL1, it will give error messages specifying the terminals and productions that are in conflict. Figure 10 shows the grammar "e-notll1.grm".

Figure 10 Grammar e-notll1.grm

```
# errors--not LL(1)
start = e .
e = e "+" t .
e = e "-" t .
e = t .
t = f "*" t .
t = f "/" t .
t = f .
f = i .
f = "(" e ")" .
```

Here’s what we get when we pass it through TCLL1:


```
Error: e is left recursive, the grammar is not LL(1)
Error: overlapping selection sets for
1.   t = f "*" t.
2.   t = f "/" t.
  overlap: {"(", i}
Error: overlapping selection sets for
1.   t = f "*" t.
2.   t = f.
  overlap: {"(", i}
Error: overlapping selection sets for
1.   t = f "/" t.
2.   t = f.
  overlap: {"(", i}
Error: overlapping selection sets for
1.   e = e "+" t.
2.   e = e "-" t.
  overlap: {"(", i}
Error: overlapping selection sets for
1.   e = e "+" t.
2.   e = t.
  overlap: {"(", i}
Error: overlapping selection sets for
1.   e = e "-" t.
2.   e = t.
  overlap: {"(", i}
7 errors and 0 warnings
```

We will consider the causes and cures of these and other problems in the subsections to follow.

4.3.2 Left-recursion removal

The error messages state that the grammar is left recursive and hence not LL(1). Why? Consider the productions:

```
e = e "+" t .
e = e "-" t .
e = t .
```

Consider the leftmost-derivation algorithm given above. When a *e* appears on top of the prediction stack, we can keep on replacing it with *e*+*t* or *e*-*t*, pushing +*t*'s and -*t*'s onto the stack and still leaving *e* on top. Eventually we will replace *e* with *t* and stop the process, but no symbol in *First(t)* will tell how many +*t*'s or -*t*'s were pushed on the stack. Similarly, while parsing, the next input symbol cannot tell us how many +'s or -'s we are going to need.

The TCLL1 parser generator checks for left recursion explicitly. The problems also appear in the reports of overlapping selection sets.

A nonterminal is directly left recursive if it occurs as the first symbol on the right hand side of one or more of its productions. If it takes more than one der-

ivation step to derive itself first, for example, $A \Rightarrow^* Bu \Rightarrow^* Cvu \Rightarrow^* Awvu$, then it is indirectly left recursive, or as we more colorfully say, there is *daisy-chain* recursion.

Direct left recursion can be removed as follows:

- Divide up the productions for the nonterminal into the left-recursive and non-left-recursive.

  $A = A\ u_1 \mid A\ u_2 \mid \ldots \mid A\ u_m \mid v_1 \mid v_2 \mid \ldots \mid v_n\ .$

- Call the $u_1\ u_2\ \ldots\ u_m$ the *tail ends* of the left recursive rules.

- Group the non-recursive right hand sides and follow them by an arbitrary repetition of the tail ends of the recursive rules, thus:

  $A = (v_1 \mid v_2 \mid \ldots \mid v_n)\ \{\ u_1 \mid u_2 \mid \ldots \mid u_m\ \}.$

When we apply this to our expression grammar, we get

```
e = t { "+" t | "-" t } .
```

TCLLk does this transformation itself. Run the grammar through tcllk, and you get:

```
>tcllk e-notll1
0 errors and 0 warnings
```

For daisy-chained left recursion, you have to first convert into direct left recursion by replacing nonterminals by their right hand sides, a technique shown below.

4.3.3 Factoring

An obvious problem for an LL(1) parser is a nonterminal having several right hand sides beginning with the same symbol. In our expression grammar, *t* has that problem:

```
t = f "*" t .
t = f "/" t .
t = f .
```

The solution is to factor the common initial part out:

```
t = f ("*" t | "/" t | ).
```

Which is to say, a *t* is an *f* followed by one of several tails.

Since one of the alternatives is empty, we can use brackets:

```
t = f  [ "*" t | "/" t ].
```

As shown above, TCLLk does its own factoring, so you don't have to. If you use the -p flag on the command line when translating grammar *e-notll1* (Figure 10 on page 26)


```
>tcllk -p e-notll1
```

you can see this resulting grammar:

```
e = t "e:101".
"e:101" = "+" t "e:101".
"e:101" = "-" t "e:101".
"e:101" =.
f = i.
f = "(" e ")".
start = e.
t = f "t:102".
"t:102" = "*" t.
"t:102" = "/" t.
"t:102" =.
```

Notice that TCLLk generates nonterminal names of the form

nonterminal : *number*

4.3.4 Replacing nonterminals by right hand sides

When faced with daisy-chained left recursion or right hand sides with conflicts but no common initial symbols to factor, we can resort to replacing nonterminals by their right hand sides to try to make the left recursion direct or the initial parts of right hands sides equal. We will call replacing nonterminals by their right hand sides “expanding” the nonterminal.

Consider the following grammar, "c-nll1.grm":

```
# c-nll1
# not LL(1)
start = s .
s = e .
s = i "=" e .
e = e "+" t .
e = e "-" t .
e = t .
t = f "*" t .
t = f "/" t .
t = f .
f = i .
f = n .
f = "(" e ")" .
```

In addition to the conflicts we have seen already in the definitions of *e* and *t*, there is a conflict between the two definitions of *s*: a string derived from *e* can also begin with an *i*.

First, let’s fix *e* and *t*:

```
e = t etail.
```



```
etail = { "+" t | "-" t } .
t = f ttail .
ttail = [ "*" t | f "/" t ].
```

Now let's start working on the production

```
s = e .
```

The *e* can derive a string beginning with an *i*. We need to rewrite until we have a production for *s* whose right hand side begins with *i* so we can factor. We replace the *e* with its one definition, giving

```
s = t etail.
```

Now we replace the *t* by its one definition

```
s = f ttail etail.
```

Now we need to replace the *f*, but it has three definitions. We must replace it with each, copying the production for each of them

```
s = i ttail etail.
s = n ttail etail.
s = "(" e ")" ttail etail.
```

Now we can factor, yielding

```
s = i ("=" e | ttail etail) .
s = n ttail etail .
s = "(" e ")" ttail etail .
```

So the resulting grammar is:

```
# c-ll1
# LL(1)
start = s .
s = i ("=" e | ttail etail) .
s = n ttail etail .
s = "(" e ")" ttail etail .
e = t etail.
etail = { "+" t | "-" t } .
t = f ttail .
ttail = [ "*" t | f "/" t ].
f = i .
f = n .
f = "(" e ")" .
```

Now you see why we created new nonterminals *etail* and *ttail*. We knew from experience that we were going to copy them in several productions, and if we left the braced or bracketed constructs in line, the parser generator would introduce multiple nonterminals with identical definitions.

TCLLk does this transformation itself, yielding:


```
e = t "e:101".
"e:101" = "+" t "e:101".
"e:101" = "-" t "e:101".
"e:101" =.
f = i.
f = "f:103".
"f:103" = n.
"f:103" = "(" e ")".
s = i "s:104".
s = "f:103" "t:102" "e:101".
"s:104" = "=" e.
"s:104" = "t:102" "e:101".
start = s.
t = f "t:102".
"t:102" = "*" t.
"t:102" = "/" t.
"t:102" =.
```

4.3.5 Daisy-chain left recursion removal

The method of expanding nonterminals in productions, replacing them with their right hand sides, is also used to remove daisy-chain, or indirect, left recursion.

When presented with this grammar:

```
start = A.
A = B x.
B = C y.
B = w.
C = A z.
```

TCLLk translates it into this grammar:

```
start = A.
A = w x "A:101".
"A:101" = z y x "A:101".
"A:101" =.
```

or if you prefer:

```
start = w x {z y x}.
```

You can check these productions by deriving a moderate-length sentence from the original grammar and examining the pattern.

You can also derive the replacement grammar yourself by first expanding B in production `A = B x.` giving `A = C y x | w x.` and then expanding `C` to get `A = A z y x | w x.`, and finally removing the direct left recursion.

4.3.6 Replacing right hand side by left hand side

If we can replace a nonterminal by all of its right hand sides, what about going the other way? Well yes, that can work, as long as we don't try to replace the definitions of the nonterminal itself. (We wouldn't want to replace $A = u \mid v$. with $A = A$.).

In fact, we have been replacing multiple right hand sides using newly created nonterminals. For example, we replaced

```
t = f "*" t .
t = f "/" t .
t = f .
```

with

```
t = f  [ "*" t | "/" t ].
```

knowing that the brackets create a new nonterminal. The translation done by the parser generator makes this explicit:

```
t = f t_3_8.
t_3_8 = "*" t.
t_3_8 = "/" t.
t_3_8 =.
```

Notice that just as replacing a nonterminal in a production required substituting each of its right hand sides, duplicating the production as necessary, the substitution the other way requires each right hand side be found at the same place in otherwise identical productions and that all those productions be replaced with a single production.

Here's a more tricky use of this technique. Suppose we have a language where statements can have any number of statement labels preceding them. The statement labels are identifiers followed by colons, and assignment statements begin with an identifier:

```
# ls-nll1
# not LL(1)
start = labeled_statement .
labeled_statement = label statement .
label = { i ":" }.
statement = i "=" e.
```

The TCLL1 parser generator will find a conflict in *label = {id ":"}.* which it reports as shown in Figure 11.

The problem is that the empty right hand side can be followed by the identifier at the beginning of the assignment statement. (The reason it's a warning rather than an error will be discussed later when talking about the "dangling else problem".

Figure 11 The warning generated for the labeled statement

```
Warning: overlapping selection sets for
     label_5_9 = i ":" label_5_9.
and empty-deriving production
     label_5_9 =.
  overlap: {i}
0 errors and 1 warning
```

Let's try rewriting *label* in *labeled_statement* to allow us to factor. First, we rewrite the definition of label to make the right recursion explicit:

```
label = i ":" label | .
```

And then replace it in *labeled_statement*:

```
labeled_statement = i ":" label statement .
labeled_statement = statement .
```

Now rewriting *statement* in *labeled_statement* = *statement* . gives

```
labeled_statement = i "=" e.
```

allowing us to factor

```
labeled_statement = i labeled_statement_tail .
labeled_statement_tail = "=" e .
labeled_statement_tail = ":" label statement .
```

If we run this through the parser generator, alas, we find the same warning. We still have *label* followed by *statement* which is the same problem as before.

But now we can apply the trick of rewriting a right hand side as its left hand side. We know we have not changed the set of strings that *labeled_statement* generates so that the strings are still described by the single right hand side *label statement*. We replace *label statement* with *labeled_statement* in the last production giving

```
labeled_statement = i labeled_statement_tail .
labeled_statement_tail = "=" e .
labeled_statement_tail = ":" labeled_statement .
```

This definition works.

4.3.7 Look-ahead trees

Unfortunately, TCLLk is not smart enough to use the transformation described in section 4.3.6, replacing right hand sides with left hand sides. Instead, it uses a look-ahead of greater than one symbol. After reading the grammar, the productions are:

```
label = label_5_9.
```



```
label_5_9 =.
label_5_9 = i ":" label_5_9.
labeled_statement = label statement.
start = labeled_statement.
statement = i "=" e.
```

The nonterminal `label_5_9` was generated for

```
label = { i ":" }.
```

After TCLLk finishes, the grammar is:

```
"label_5_9:101" = i "label_5_9:102".
"label_5_9:102" = "=" "label_5_9:103".
"label_5_9:102" = ":" "label_5_9:101".
"label_5_9:103" = BACKUP BACKUP.
labeled_statement = "label_5_9:101" statement.
start = labeled_statement.
statement = i "=" e.
```

BACKUP is an action symbol that tells the parser to back up one symbol in the input.

Nonterminal `"label_5_9:101"` recognizes sequences of labels of the form `i ":"`, i.e. `{ i ":" }`. Nonterminal `"label_5_9:101"` can be followed by a statement of the form `i "=" e`.

Production

```
"label_5_9:101" = i "label_5_9:102".
```

says to find a `{ i ":" }` which can be followed by `i "=" e`, first look for an `i` and then look for a `"label_5_9:102"` . The `i` could be the `i` in `i ":"` or the `i` in `i "=" e` .

Productions

```
"label_5_9:102" = "=" "label_5_9:103".
"label_5_9:102" = ":" "label_5_9:104".
```

say that to find a "label_5_9:102" first find either a `"="` or a `":"`.

Finding a "=" look for a "label_5_9:103" next. Having seen a "=" , we now know that we are looking at a statement of the form `i "=" e`. We have already read past the `i` and the `"="` . Production

```
"label_5_9:103" = BACKUP BACKUP.
```

says to back up two positions, pushing the `i "="` back into the input. Since there is no other nonterminal to look for, we are done finding a `"label_5_9:103"` , a `"label_5_9:102"` , and a `"label_5_9:101"`. We started looking for `"label_5_9:101"` in the production


```
labeled_statement = "label_5_9:101" statement.
```

so we expect to find a statement next. So now we are looking for a statement given by production

```
statement = i "=" e
```

This is exactly what we want. We are looking for `i "=" e` with the `i "="` back on the input.

Now consider the other production

```
"label_5_9:102" = ":" "label_5_9:101".
```

It says that having just found an `i`, look for a `":"` next and look for a `"label_5_9:101"` after that. This is a normal right recursive definition of `{ i ":" }`. Actually, it is optimized from these productions:

```
"label_5_9:102" = ":" "label_5_9:104".
"label_5_9:104" = BACKUP BACKUP i ":"
      "label_5_9:101".
```

which says that we've looked ahead and discovered we are finding a `{ i ":" }` . We back up two symbols, pushing the `i ":"` back on the input and then look for the production that represents `{ i ":" }` , i.e.

```
label_5_9 = i ":" label_5_9.
```

in the original grammar.

But the two `BACKUP` actions would put the `i ":"` back on the stack, and then the `i ":"` would match them again. So the two BACKUPs and the two terminals cancel each other out.

4.4 Tables of operators

You may be given tables of operators with their precedences and associativities. You may have to translate these into context free syntax. This is so easy, TCLLk won't help you with it.

Before looking at the algorithm for converting operator precedence tables into productions, consider this example. Given the table of operators, Table 1.

Table 1 Precedence table.

Operators	**Unary or binary**	**Associativity**	**Precedence**
%	unary	many	highest
$!	binary	left	
&	binary	right	

Table 1 Precedence table.

#	binary	non-associative	
^	unary	one-at-most	lowest

the algorithm will give the following grammar:

```
E1 = ^ E2 .
E1 = E2 .
E2 = E3 # E3 .
E2 = E3 .
E3 = E4 & E3 .
E3 = E4 .
E4 = E4 $ E5 .
E4 = E4 ! E5 .
E4 = E5 .
E5 = % E5 .
E5 = E6 .
E6 = F .
```

Binary operators associate to the left if the left hand side nonterminal is left recursive, and associate to the right if the nonterminal is right recursive. The higher precedence operator must occur in a subphrase of the lower precedence one.

Here is the algorithm for generating productions from a precedence table of binary operators:

Suppose the grammar specifies

E = E P E .

E = F

with tables giving the precedence and associativity of the operators, *P*.

Number the precedence levels consecutively, *1, 2, ..., n* from lowest to highest.

Create a nonterminal, E_i, for $1 \le i \le n+1$.

Create a renaming production

$$E_i = E_{i+1} .$$

for all $i \le n$.

For each binary operator P_j at precedence level *i*, if P_j is left associative, put in a production

$$E_i = E_i \ P_j \ E_{i+1} .$$

If P_j is right associative, put in a production

E_i = E_{i+1} P_j E_i .

Or if P_j is non-associative (for example, the relational operators in Pascal), put in a production

E_i = E_{i+1} P_j E_{i+1} .

If you end up with two productions

E_i = E_i ... .

and

E_i = ... E_i .

you have generated an ambiguous grammar; left and right associative operators must not be at the same precedence.

For each unary operator *P* at precedence level *i*, if several occurrences of *P* may occur in a row, put in a production

E_i = *P* E_i .

Or, if at most one occurrence of *P* can occur in front of an operand, put in the production

E_i = *P* E_{i+1} .

Add productions

E = E_1 .
E_{n+1} = F .

4.5 The dangling-else problem

The dangling-else problem occurs in languages that have optional *else* clauses in *if* statements and no *if* statement terminator (such as *end if* or *fi*). In nested *if*s, it is not clear which preceding *if* an *else* goes with. This ambiguity shows up when we try to construct an LL(k) parser. Consider the syntax:

```
statement = if_statement
     | i "=" e.
if_statement = if e then statement
     | if e then statement else statement .
```

this obviously will have a conflict. So we try factoring

```
if_statement = if e then statement else_option.
else_option = [ else statement ].
```

When we pass this through the TCLL1 parser generator, we get these warnings:


```
Warning: overlapping selection sets for
     else_option_6_15 = else statement.
and empty-deriving production
     else_option_6_15 =.
  overlap: {else}
0 errors and 1 warning
```

Why? First consider this example:

```
if e then if e then if e then i=e else i=e else
i=e
```

We have three **if**'s and two **else**'s. Which **else** goes with which **if**? When the LL(1) recognizer has just finished processing the first *i=e*, there will be three *else_option*'s on the prediction shack. Two of them must be replaced with *else statement*; one, with the empty string. Which?

Observe that a *statement* can be followed by an *else_option* and an *if_statement* can end in an *else_option*. The *else_option* at the end of an *if_statement* can therefore be followed by an *else*, which means that it is unclear how to choose between a right hand side beginning with an *else* and the empty right hand side.

Unfortunately, there's no way to get rid of this problem. (Well, if you are the language designer, you could redesign the language, but if you are only the compiler writer, you have to take the language as given.) So here's what we do: we cheat. We want the parser to associate the *else* with the innermost *if*. This will be the *if* statement that placed the *else_option* on top of the prediction stack. So we let the *else_option* on top of the prediction stack handle the *else*. That means we will choose the right hand side that has the *else* in its first set rather than the right hand side that is empty and only has the *else* in its follow set.

Why don't we just look for if and else and treat them specially? Actually, the problem occurs in more language constructs than if statements. In general we can call it the "dangling tail" problem. To handle the general dangling tail problem, both the TCLL1 and TCLLk parser generators use this rule:

> Only use a symbol from the follow set to choose an empty-deriving right hand side if it does not appear in the first set of any right hand side.

The parser generator fills a table, *sel*, that maps nonterminals and terminals into right hand sides. For nonterminal *A* and terminal *t*, *sel[A,t]* is the right hand side to replace *A* with if *t* is next in the input.

The parser generator works in two passes over a nonterminal's productions:

1. For each production *A = u.* and every terminal *t* in *First(u)*, *sel[A,t]* is assigned right hand side *u*. If TCLLk finds that sel[A,t] already has been assigned a different right hand side, it reports an error.
2. Then TCLLk checks to see if there is a production for *A* with an empty-deriving right hand side, *i.e., A = w.* where *w* is either empty or is com-

posed of action symbols and of nonterminals each of which derives the empty string. If there is no such right hand side, TCLLk is done with this nonterminal. If there are two or more such productions, the grammar is ambiguous—there is more than one way to derive the empty string from *A*. If there is precisely one such production, *A = w.*, then for all symbols *t* in *Follow(A)*,

- if *sel[A,t]* already has a right hand side assigned, issue a warning,
- otherwise assign *sel[A,t]* the right hand side *w*.

We have the TCLL1 parser generator give a warning when first sets and follow sets give conflicting choices for a nonterminal since it may not be a dangling else problem. Indeed, in the labeled statement example, it wasn't. If we'd used the parser that was generated with this warning, it would never have been able to parse an assignment statement: it would assume an identifier at the beginning of a statement had to be a label and it would report an error when it saw an "=" rather than a ":".

What about TCLLk? It handles dangling tails without even giving a warning.

Chapter 5 Parsing with action symbols

We have talked about grammars being used to *derive* sentences from the start symbol by replacing nonterminal symbols by their right hand sides, but this is just the reverse of what we need for parsing; we need to *reduce* the sentence to the start symbol by repeatedly replacing right hand sides by their left hand sides. As each reduction is made, a semantic value is computed for the left hand side symbol from the semantic values of the right hand sides. The procedures that compute these values are called semantics routines or action routines. In addition to computing semantic values, the semantics routines can also access shared data structures and write to files. In theory, “the meaning of the program" is the semantic value assigned to the start symbol. In practice it can be the contents of a data structure or the contents of a file.

5.1 Reductions

If we start with a sentence and just look through the right hand side for substrings we can reduce, we may go down blind alleys and never reduce it to the start symbol. Using our expression grammar, we could try the following reduction sequence on *i*i*:

```
i * i
f * i
t * i
e * i
e * f
e * t
e * e
```

whereupon we cannot make any further reductions.

There are some parsing algorithms, called *bottom-up parsing algorithms*, that find the correct substring to reduce each step. These parsing algorithms can be used directly. Unfortunately, LL(1) parsing is *top-down*, so we must do something to make it give us the reduction sequence.

Here’s what we do: We invent an extension of the language in which each production ends with a distinct terminal symbol, a *marker*, translate the sentence into the equivalent *marked* sentence in this extended language, and use the marked sentence to compute the semantic associations for the nonterminals. We will show that

- we can use the markers to perform reductions in the correct order.

- we can translate a sentence without markers into a sentence with markers using a version of the LL(1) recognition algorithm.
- we can combine these two operations so that no intermediate sentence is ever generated.

First, let's consider how we would use markers for reductions. We add markers to our expression grammar to give an marker-augmented grammar as shown in Table 2.

Table 2 Productions in grammar and grammar with markers.

Original grammar	Marked grammar
`start = e.`	`start = e P1.`
`e = e "+" t.`	`e = e "+" t P2.`
`e = e "-" t.`	`e = e "-" t P3.`
`e = t .`	`e = t P4 .`
`t = f "*" t.`	`t = f "*" t P5.`
`t = f "/" t.`	`t = f "/" t P6.`
`t = f .`	`t = f P7 .`
`f = i.`	`f = i P8.`
`f = "(" e ")".`	`f = "(" e ")" P9.`

Notice that each production in the original grammar has a corresponding production in the marked grammar. The only difference between these productions is that the marked production has a marker at the end of its right hand side. All the markers are distinct.

Assuming *i* represents *integer*, here's a sentence in the expression language and its translation:

```
30 / 5 * 2 + 6
30 P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
```

The corresponding sentences can be derived by leftmost derivations using the corresponding productions in each derivation step, as shown in Table 3.

Table 3 Parallel derivation of sentence and sentence with markers.

start	start
e	e P1
e + t	e + t P2 P1
t + t	t P4 + t P2 P1
f / t + t	f / t P6 P4 + t P2 P1

Table 3 Parallel derivation of sentence and sentence with markers.

30 / t + t	30 P8 / t P6 P4 + t P2 P1
30 / f * t + t	30 P8 / f * t P5 P6 P4 + t P2 P1
30 / 5 * t + t	30 P8 / 5 P8 * t P5 P6 P4 + t P2 P1
30 / 5 * f + t	30 P8 / 5 P8 * f P7 P5 P6 P4 + t P2 P1
30 / 5 * 2 + t	30 P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + t P2 P1
30 / 5 * 2 + f	30 P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + f P7 P2 P1
30 / 5 * 2 + 6	30 P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1

When reducing the translated sentence, we use the markers as suffix Polish operators. Each marker has a number of symbols preceding it in its right hand side, which is the number of operands it takes as a suffix Polish operator. The numbers for the markers are shown below:

marker	P1	P2	P3	P4	P5	P6	P7	P8	P9
number of operands	1	3	3	1	3	3	1	1	3

Now we will show how to reduce a marked sentence to the start symbol. The input consists of a string of tokens and markers. The algorithm uses a stack.

Figure 12 Algorithm for Reduction using markers.

ALGORITHM FOR REDUCTION USING MARKERS

Initially set the stack empty.

Read through the marked sentence one symbol at a time

if the symbol is a token, push it on the stack

otherwise the symbol is an marker,

look up the production it occurs in

remove the marker's "*arity*" number of symbols from the stack (these correspond to the symbols ahead of the marker on the right hand side)

push the left hand side symbol on the stack

At the end, the start symbol will be on the stack.

The markers coming at the ends of right hand sides tell us when to make a reduction and which production to use. A reduction sequence using the algorithms shown in Table 4 on page 44.

Table 4 Reduction sequence of expression with markers.

stack	input
	30 P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
30	P8 / 5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f	/ 5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f /	5 P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / 5	P8 * 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / f	* 2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / f *	2 P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / f * 2	P8 P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / f * f	P7 P5 P6 P4 + 6 P8 P7 P2 P1
f / f * t	P5 P6 P4 + 6 P8 P7 P2 P1
f / t	P6 P4 + 6 P8 P7 P2 P1
t	P4 + 6 P8 P7 P2 P1
e	+ 6 P8 P7 P2 P1
e +	6 P8 P7 P2 P1
e + 6	P8 P7 P2 P1
e + f	P7 P2 P1
e + t	P2 P1
e	P1
start	

5.2 Semantic values

Every symbol in the sentential form has a meaning associated with it, a *semantic value*. The semantic values of symbols are also called collections of *attributes*. Terminal symbols will have semantic values assigned to them by the scanner. Terminal symbols with their associated values are called *tokens*. In the TCLLk system, a token is a record containing

- the syntactic type (the terminal symbol)— used by the parser to recognize the input.
- the body (the string of characters that comprise the token)—used by the se-

mantics routines.

- the line number on which the token occurred.
- the column number (actually the character position) of the leftmost character of the token; the line and column are used to report the position of an error.

It is fairly clear how to use the reduction algorithm to compute semantic values of symbols. Each production, and hence each marker, has a procedure, a *semantics routine*, associated with it. What is kept on the stack are semantic values. When a marker is encountered, the semantic values of the right hand side symbols are removed from the stack and passed to the semantics routine. The routine computes the semantic value of the left hand side symbol and that value is pushed back on the stack.

A semantic value of a nonterminal expresses the meaning of the phrase it derived. A semantic value may be:

- The numeric value of the subexpression the nonterminal represents.
- An operator tree or an abstract syntax tree representing the phrase.
- A translation of the phrase and a description of its result's data type.

5.3 Inserting markers into sentences

So how do we insert markers into a sentence? We use a version of our LL(1) recognition algorithm. The differences from the original recognition algorithm are as follows:

- The algorithm uses a grammar containing markers.
- As it matches tokens, it writes them out.
- When it finds a marker on the top of the prediction stack, it writes it out.

The LL(1) translation algorithm with action symbols is shown in Figure 13.

Of course a grammar has to be put in LL(1) form before the parser can use it. Do markers cause any problems? Not really. All they require is:

- Markers are moved around like any other symbol.
- When calculating First and Follow sets, markers are invisible; they are treated like nonterminals that derive only the empty string.

If we transform the marked expression grammar, we can get the following LL(1) form:

Table 5 Expression grammar with markers in LL(1) form.

```
start = e P1.
e = t P4 etail.
```


Table 5 Expression grammar with markers in LL(1) form.

```
etail = "+" t P2
etail.
etail = "-" t P3
etail.
etail =.
t = f ttail .
ttail = "*" t P5.
ttail = "/" t P6.
ttail = P7 .
f = i P8.
f = "(" e ")" P9.
```

Figure 13 LL(1) Algorithm to insert markers

LL(1) ALGORITHM TO TRANSLATE
INTO A MARKED SENTENCE

Initially, place the start symbol and the EOI (end of input) symbol on the prediction stack with the start symbol on top. Put the EOI symbol at the end of the input. Read the first input symbol into the *current token*.

Repeat

pop the *top symbol* off the prediction stack.

if the *top symbol* is a marker, write it out.

otherwise if the *top symbol* is a terminal, compare it to the current token.

If they match, write the *current token* out and read the next token from the input into the *current token*.

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

otherwise if the *top symbol* is a nonterminal, choose one of its right hand sides and push it on the prediction stack, leftmost symbol on top. Choose the right hand side by looking at the next input symbol and deciding which RHS will allow parsing to continue.

until the EOI symbol is matched.

5.4 Parsing

In practical parsers, we do not first insert markers into a sentence and then pass it through a reduction algorithm. We combine both parts in one algorithm.

In the following LL(1) parsing algorithm, we use the name *action symbols* for *markers*. When the parser sees an action symbol, it calls an *action routine*, sometimes called a *semantics routine*. Is there a difference between a marker and an action symbol? Well, yes. All markers are action symbols, but we can put in action symbols for other purposes than marking the end of a right hand side, *e.g.* putting the scanner into a different mode.

5.4.1 Action symbols

In TCLLk, action symbols are required to be identifiers; they are used as names of the procedures used for action routines. Action symbols may be declared by following them with an exclamation point, *e.g.*

```
f = "(" e ")" P9! .
```

If the action symbol has been declared in one place with an exclamation point, it need not be followed by an exclamation point anywhere else.

If you don't care to use the exclamation point, you can declare action symbols with the following declaration:

actions : a_1 a_2 … a_n **.**

where each a_i is an action symbol.

5.4.2 The LL(1) parsing algorithm

The LL(1) parsing algorithm with action symbols is shown in Figure 14.

A bit of explanation is necessary about marking the current token present or absent. In earlier algorithms we read the first token at the beginning and then read in a new token as soon as we had recognized the previous. This is quite all right for some compilers, but it is particularly a problem for interactive programs. The system won't respond to one command until it has seen the first token of the next. Here we don't try reading another token until we are going to look at it. We can perform any number of actions after recognizing a token before requesting the next, allowing the program to respond immediately after the last token of the command has been read. Both the TCLL1 and TCLLk use this algorithm.

5.4.3 Building parsers

Here is an approach for building parsers:

First, design a grammar for the language which has meaningful phrases. It must be clear to you what action you wish to take at the end of each phrase and what the semantic value of each symbol in the grammar is. Each token *is* a semantic value (the value of the terminal symbol). Each nonterminal has an associated data type to contain its semantic value or attributes.

Put an action symbol at the end of the right hand side of each production. Each production has some rule for constructing its left hand side's semantic value from the semantic values of the right hand side symbols (in addition to writing

Figure 14 LL(1) parsing algorithm.

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 some 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 next input symbol and deciding which right hand side will allow parsing to continue.

until the EOI symbol is matched.

out translated code and changing some global variables). The action symbol is the name of the procedure to call when that right hand side has been recognized. It will pull off the semantics stack one value for each symbol on the right hand side and will push back the value of the left hand side.

Several productions may have the same action symbol if the number of elements on the right hand side are the same and the actions are similar. For example, each binary operator could have its own action routine, or all binary operators could share the same routine that looks at the operator token to decide what to do.

You may omit an action symbol for a renaming production, a production that has exactly one symbol on the right hand side and no action except to push back the value it pops. You may introduce action symbols at other places than the ends of right hand sides if you feel the need; not all action symbols represent markers.

Give the grammar to TCLLk. When TCLLk transforms the grammar to LL(1) form (perhaps with look-aheads and BACKUP actions), it will move around action symbols the same as any other symbol. When checking First and Follow

sets, it will treat action symbols as if they are nonterminals that derive only the empty string. Alas, if TCLLk can't build tables for the grammar, you will have to try some transformations yourself. You might not be successful. Not all languages are LL(k).

Write the action routines. An action routine for a *marker* action symbol will pull values off the semantic stack for the right hand side symbols of a production, compute the semantic value of the left hand side, and push it back. However, an action routine that does not correspond to a marker is not required to pop any value off the semantics stack or push a value back. You may also decide that some nonterminals have no semantic value and hence do not need to have a value on the semantics stack. Feel free not to push a value for such a symbol, but be aware that it will complicate keeping track of the semantics stack's depth, as we will discuss later.

5.4.4 Example of evaluating arithmetic expressions

Let's design action routines to evaluate arithmetic expressions using our expression grammar. Here's a sentence in the language:

30/5*2+6

Suppose we parse it using the LL(1) grammar with markers we constructed before:

Table 6 Expression grammar with markers at ends of phrases.

```
start = e P1.
e = t P4 etail.
etail = "+" t P2 etail.
etail = "-" t P3 etail.
etail =.
t = f ttail .
```

```
ttail = "*" t P5.
ttail = "/" t P6.
ttail = P7 .
f = i P8.
f = "(" e ")" P9.
```

In this case the terminal symbol *i* represents an integer token. Here's what the action routines are expected to do:

- **P1** pop the numeric value on top of the semantics stack, write it out, and terminate execution.
- **P2** pop three values from the semantics stack, add the first and third, and push the sum.
- **P3** pop three values from the semantics stack in order z, y, x; push the value x-z back on the stack.
- **P4** no operation.
- **P5** pop three values from the semantics stack in order z, y, x; push the value x*z back on the stack.

P6 pop three values from the semantics stack in order z, y, x; push the value x/z back on the stack.

P7 no operation.

P8 pop the token off the semantics stack, convert its body from a string to an integer, and push the value back.

P9 pop three values off the semantics stack and push the middle value back.

Here's a trace of the input and the semantics stack while parsing the sentence *30/5*2+6*. Tokens are indicated as *type:value*.

Table 7 Trace of a parse.

Action from previous	Semantics stack	Prediction stack	Input
		start EOI	30/5*2+6 EOI
		e P1 EOI	30/5*2+6 EOI
		t P4 etail P1 EOI	30/5*2+6 EOI
		f ttail P4 etail P1 EOI	30/5*2+6 EOI
		i P8 ttail P4 etail P1 EOI	30/5*2+6 EOI
match	i:30	P8 ttail P4 etail P1 EOI	/5*2+6 EOI
P8	30	ttail P4 etail P1 EOI	/5*2+6 EOI
	30	"/" t P6.P4 etail P1 EOI	/5*2+6 EOI
match	30 /:/	t P6 P4 etail P1 EOI	5*2+6 EOI
	30 /:/	f ttail P6 P4 etail P1 EOI	5*2+6 EOI
	30 /:/	i P8 ttail P6 P4 etail P1 EOI	5*2+6 EOI
match	30 /:/ i:5	P8 ttail P6 P4 etail P1 EOI	*2+6 EOI
P8	30 /:/ 5	ttail P6 P4 etail P1 EOI	*2+6 EOI
	30 /:/ 5	"*" t P5 P6 P4 etail P1 EOI	*2+6 EOI
match	30 /:/ 5 *:*	t P5 P6 P4 etail P1 EOI	2+6 EOI
	30 /:/ 5 *:*	f ttail P5 P6 P4 etail P1 EOI	2+6 EOI
	30 /:/ 5 *:*	i P8 ttail P5 P6 P4 etail P1 EOI	2+6 EOI
match	30 /:/ 5 *:* i:2	P8 ttail P5 P6 P4 etail P1 EOI	+6 EOI
P8	30 /:/ 5 *:* 2	ttail P5 P6 P4 etail P1 EOI	+6 EOI
	30 /:/ 5 *:* 2	P7 P5 P6 P4 etail P1 EOI	+6 EOI

Table 7 Trace of a parse.

P7	30 /:/ 5 *:* 2	P5 P6 P4 etail P1 EOI	+6 EOI
P5	30 /:/ 10	P6 P4 etail P1 EOI	+6 EOI
P6	3	P4 etail P1 EOI	+6 EOI
P4	3	etail P1 EOI	+6 EOI
	3	"+" t P2 etail P1 EOI	+6 EOI
match	3 +:+	t P2 etail P1 EOI	6 EOI
	3 +:+	f ttail P2 etail P1 EOI	6 EOI
	3 +:+	i P8 ttail P2 etail P1 EOI	6 EOI
match	3 +:+ i:6	P8 ttail P2 etail P1 EOI	EOI
P8	3 +:+ 6	ttail P2 etail P1 EOI	EOI
	3 +:+ 6	P7 P2 etail P1 EOI	EOI
P7	3 +:+ 6	P2 etail P1 EOI	EOI
P2	9	etail P1 EOI	EOI
	9	P1 EOI	EOI
P1		EOI	EOI
match			

5.5 Accounting for semantics stack depth[1]

As mentioned, an action routine can push either one or zero values on the semantics stack. As a rule, they would leave one value to represent the left hand side symbol. Some nonterminals, however, have no semantic information associated with them, so there is no reason to keep a value on the stack for them. It is a strong temptation not to needlessly push and pop null values, and we are sure to give in to this temptation, but it makes it harder to get our parser right. We will probably find one of our biggest problems with this parsing method is that we mangle the semantics stack by popping or pushing the wrong number of items.

1. *This section was designed for TCLL1, where you would transform the grammar into LL(1) form by hand and then have to maintain it in its LL(1) form. Some of this is probably useful for TCLLk, but it needs rewriting. Feel free to ignore this section.*

The symptom that you have mangled the semantics stack is semantics routines crashing, complaining that they have the wrong types of operands or that they are trying to pop values off an empty semantics stack.

Recall that the paradigmatic way to use action symbols involves four things:

1. Write an original grammar in a clear, meaningful form without using any grouping, optional, or repetitive constructs and with action symbols only at the ends of right hand sides.

2. Design the action routines to remove one thing from the semantics stack for each symbol ahead of them on the right hand side and will push one value back.

3. Then, create a transformed grammar in LL(1) form, moving the action symbols around like any other symbol. This assumes, of course, you are transforming it by hand and using TCLL1. You won't be doing this, probably, if you are using TCLLk.

4. Represent every terminal and nonterminal symbol in the original grammar by exactly one value on the semantics stack.

If we decide not to push values for some nonterminals, you will have to keep track of which nonterminals have values and which do not. It will no longer be immediately obvious by looking at a right hand side just how many values an action symbol's procedure is to pop or push. If you put action symbols in front of or in the middle of productions, it makes it harder to figure out what's going on. And using braces, brackets, parentheses, and vertical bars causes more confusion.

The problem is made all the worse if you transform the grammar into LL(1) form by hand. When you need to make a change in the grammar (and you will) you will make the change directly to the LL(1) form and it will not be at all clear what effect it will have on the semantics stack. In the LL(1) form, newly introduced nonterminals will not necessarily leave either zero or one values on the stack.

What we will need is a way to account for stack depth. Associate a number with each symbol, right hand side, alternative, and parenthesized, optional, or repetitive phrase. These numbers represent the effect of the construct on the semantics stack depth.The rules are shown in Figure 15 on page 53.

To use a version of our expression grammar:

```
start = e P1!.
e = t { "+" t P2! | "-" t P3!} .
t = f  [ "*" t P5! | "/" t P6!].
f = i P8! | "(" e ")" P9!.
```


Figure 15 Rules for accounting for semantic stack depth.

1. Every symbol will change the depth of the semantics stack by a fixed amount.
   - All terminals count as +1. The parser will push each token matched on the stack.
   - Each nonterminal will have a fixed number of symbols it will leave on or remove from the stack. Nonterminals in the original grammar will change the stack depth by +1 or +0. (Nonterminals introduced during the translation to LL(1) form may even have a negative net depth.)
   - An action symbol has an effect equal to the number of symbols pushed minus the number popped. Since the number pushed is typically zero or one and the number popped is greater than or equal to zero, an action symbol can have any number less than or equal to one.
2. A string of symbols has a number computed by adding up all its components.
3. The number associated with a nonterminal must be the same as the number computed for each of its right hand sides.
4. Each alternative (separated by vertical bars, |) must add up to the same value.
5. The contents of brackets, [...], must add up to zero.
6. The contents of braces, {...}, must add up to zero.

We can determine the numbers associated with the symbols as shown in Table 8.

Table 8 Semantics stack depth changes by symbols.

"+", "-", "*", "/", i, "(", ")"	1	they all are terminals
start	0	
e, t, f	1	they are nonterminals from the original grammar
P2, P3, P5, P6	-2	they handle binary expressions, popping three and pushing one
P1	-1	it pops an expression’s value and pushes nothing
P8	0	it pops the integer token and pushes its numeric value

Table 8 Semantics stack depth changes by symbols.

P9	-2	it pops three values and pushes back the middle one

We can now compute the lengths of the right hand sides to make sure the rules aren't violated and the lengths of the left hand sides match. A rough trace of the calculations we may need to go through is shown in Table 9.

Table 9 Testing semantics stack depth changes.

syntax	calculation	number of the rule being used or checked, from Figure 15
`start = e P1!.`	0 = 1 + -1	3
`"+" t P2!`	1 + 1 + -2 = 0	2
`"-" t P3!`	1 + 1 + -2 = 0	2
`"+" t P2! | "-" t P3!`	0 = 0	4
`{ "+" t P2! | "-" t P3!}`	0	6
`e = t { "+" t P2! | "-" t P3!}`	1 = 1 + 0	3
`"*" t P5!`	1 + 1 + -2 = 0	2
`"/" t P6!`	1 + 1 + -2 = 0	2
`"*" t P5! | "/" t P6!`	0 = 0	4
`[ "*" t P5! | "/" t P6!]`	0	5
`t = f  [ "*" t P5! | "/" t P6!].`	1 = 1 + 0	3
`i P8!`	1 + 0 = 1	2
`"(" e ")" P9!.`	1 + 1 + 1 + -2 = 1	2
`i P8! | "(" e ")" P9!`	1 = 1	4
`f = i P8! | "(" e ")" P9!`	1 = 1	3

Chapter 6 Panic mode error recovery

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. What should it do?

First, of course, the parser should give an error message. The easiest error message is simply:

unexpected token XXXX at line YYYY, column ZZZZ

Then what? Just stopping isn't nice. Users appreciate the compiler trying to find several errors with each attempted compile. The compiler should attempt to recover from the error and continue processing the program.

There are two problems in attempting to continue:

- 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.

Fortunately, both are easy to accomplish with LL parsing.

A simple error recovery technique for LL parsers is called *panic mode*. When the parser has detected and reported an error, it goes into panic mode and throws away part of the input and part of the prediction stack until it has found a token in the input and a symbol in the prediction stack that allow parsing to continue, then it returns to normal mode and continues parsing.

How does it choose an input symbol to restart at, and how does it decide how much of the stack to throw 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 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. You must specify the others yourself with the fiducials declaration:

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

Notice that the declaration uses a colon rather than an equal sign, the fiducials are listed without commas and the declaration concludes with a period.

“But,” you may ask, “if the parser just throws away part of the prediction stack, won’t the semantics stack will be mangled when the parsing resumes. What does the parser do about that?”

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 the 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.

Chapter 7 Incorporating the parsers into compilers in Icon

Here is what you need to do to build a compiler in the Icon programming language using this system:

- create a grammar for the language you wish to compile, put in action symbols and run it through TCLLk to get tables for your parser. If your grammar is called *yourlang.grm*, the tables will be given the name *yourlang.llk*.
- Write a main program to initialize the compiler and call the parser. Actually, you will just edit an old main program to adapt it. We'll see one later that we can start with.
- Write a scanner for the language. Again, we will just adapt an already written scanner. I (TC) usually start with one written for Oberon-2. We'll see it later and see how it works. (Other compiler-writing systems provide scanner generators, but scanners are so trivial, it doesn't seem worth while.)
- Write action routines. Most of these need to be written specially for each compiler, but there is some standard boilerplate that they share.
- Compile our files together and link with *readllk*, *parsellk*, *semstk*, and *rpterr* from the TCLLk run time library and with files *xcode*, *options*, *pathfind*, *escape*, and *ebcdic* from the Icon programming library.

The call-structure of the compiler is as follows:

Our **main** program calls

- **readLLk** in file *readllk.icn* to read in the parse tables from a file and produce an internal form of the tables for the parser to use.
- **initSemanticsStack** in file *semstk.icn* to initialize the semantics stack for the action routines.
- **initScanner**, which we provide to initialize the scanner. It is used mainly to open the user's input file. We can leave this routine out if we don't need it.
- **parseLLk** in file **parsellk.icn** to read and parse the input program. Procedure *parseLLk* calls
  - **scan**, which we provide, to return it the next token of the input each

time it is called. When the input is finished, scan will return an EOI token for each call.

- **outToken** in file *semstk.icn* to put a token it has matched onto the semantics stack.
- **outError** in file *semstk.icn* to push an error token on the semantics stack during panic mode error recovery.
- **reportParseError** in file *rptperr.icn* to report the parser has encountered an unexpected token in the input.
- **outAction** in file *semstk.icn* to call an **action routine**, which you supply.
- Your **action routine** may call
  - **popSem** in file *semstk.icn* to pop a number of values off the semantics stack and return them in a list. The leftmost value in the list corresponds to the leftmost symbol in the right hand side that contains the action symbol, and is the value that was furthest down the semantics stack.
  - **pushSem** in file *semstk.icn* to push the semantics value of the left hand side symbol onto the semantics stack.
  - **anyError** in file *semstk.icn* to look through a list of values and succeed returning any of those values that is an error token, or fail if there are no error tokens present.
  - **isError** in file *semstk.icn* to check whether a particular semantic value is an error token.

7.1 Interface to readllk.icn

The TCLLk parser generator creates a file containing LL(1) parse tables for a grammar. This parse table must be read in before the parser can use it. Module *readLLk.icn* provides the routine, *readLLk*, to read in a parse table. Routine *readLLk* returns the parse table contained in a record of type *LLk*.

record LLk(...)

We don't need to know the fields of this record to use the parser. Procedure *readLLk* returns a record of this type; procedure *parseLLk* takes it as a parameter.

procedure readLLk(fileName)

parameter: fileName—a string, the name of the file containing the output of the TCLLk parser generator.

returns a record of type LLk containing parse tables

fails if it can’t open file fileName

Procedure *readLLk* takes the name of the parse table file as a string. (TCLLk creates the file with the extension ".LLk" so unless you’ve renamed it, you will pass a file name with that extension.) If it successfully reads the tables, *readLLk* will return a record of type *LLk* containing an internal form of the tables. If it can’t open the file, readLLk will fail. Unfortunately, if the file is malformed, the Icon library routine xdecode will fail.

7.2 Interface to parsellk.icn

Module *parsellk.icn* contains the parser and the record declaration for tokens, the record *Token*. The scanner returns a token to the parser for each input symbol. Tokens are pushed on the semantics stack as they are recognized.

record Token(type,body,line,column)

fields:

1. type—a character string, the identifier or string used in the grammar to represent the terminal symbol.
2. body—the character string that the scanner found in the input. For keywords and most punctuation, the bodies will usually be the same as the type. For identifiers, the body will be the name of the identifier. For constants, the type will indicate the type of the constant and the body will have the character string the user wrote.
3. line—an integer, the line number where the token was found.
4. column—an integer, the character position of the token in the line (tabs are treated as single characters).

If we are allowing "includes" you may want to add another field to tell which file the token was found in.

procedure parseLLk(LLk)

parameter: LLk—a record of type *LLk*

returns nothing

Procedure parseLLk performs an entire parse up to the end of input. It must be given an *LLk* record containing the parse tables. (See module *readLLk.icn* for a further discussion of record *LLk* and procedure *readLLk* to read in the tables.)

7.3 Interface to semstk.icn

Module *semstk.icn* provides procedures to maintain the semantics stack. The parser uses three of the routines; we use the rest. This module provides the definition of record *ErrorToken*, which has exactly the same fields as *Token*, but is used to represent erroneous phrases.

record ErrorToken(type,body,line,column)

The parser inserts error tokens during panic mode error recovery. Our action routines should check for error tokens before taking any action. Once either the parser or an action routine has reported an error, error tokens should be pushed on the semantics stack to warn other action routines not to give another error message and not to try to make sense of the input.

procedure initSemanticsStack()

called by our main program

parameters: none

returns nothing

This procedure should be called by the main program before starting parsing. As its name says, it initializes the semantics stack.

procedure outToken(tok)

called by the parser

parameter: tok—a token

returns nothing

The parser calls procedure *outToken* to push a token on the semantics stack.

procedure outAction(a)

called by the parser

parameter: a—a string, an action symbol, the name of an action routine.

returns nothing

The parser calls procedure *outAction* to call an action routine. The parser passes *outAction* the string name of the routine to call.

procedure outError(t,l,c)

called by the parser

parameters:

t—a string, the name of a terminal symbol

l—an integer, a line number

c—an integer, a position on the line

returns nothing

The parser calls procedure *outError* to push an error token on the stack. The error token will have the type and body *t*, line *l* and column *c*.

procedure popSem(n)

called by an action routine

parameter: n—an integer, the number of values to pop from the semantics stack

returns a list containing the values popped, topmost at the right

We call procedure *popSem* to remove the top *n* values from the semantics stack and return them to us in a list. The top element will be the rightmost value in the list. Say we call this from an action routine *A* and the grammar has a production:

$L = R_1\ R_2\ ...\ R_k\ A!.$

where each symbol R_i has a value V_i on the semantics stack, then

popSem(k)

will yield a list

$[V_1, V_2, ..., V_k]$

procedure pushSem(s)

called by an action routine

parameter: s—a value to push on the semantics stack

returns nothing

We call procedure *pushSem* to push a value on the semantics stack.

procedure isError(v)

called by an action routine

parameter: v—a value, presumably from the semantics stack

returns: an undefined value if v is an ErrorToken record

fails if v is not an error token

Procedure *isError* will succeed if *v* is an ErrorToken record and will fail otherwise.

procedure anyError(V)

called by an action routine

parameter: V—a list of values, presumably from the semantics stack

returns: an ErrorToken record, v, found in the list V if there is any

fails if V does not contain any error tokens

Procedure *anyError* looks through list *V* to see if it contains any error tokens. If *V* does, then *anyError* will succeed returning one of the error tokens in *V*. If there are no error tokens, then *anyError* fails.

7.4 Interface to action routines

We will need to provide an action routine for each action symbol. The routine has the same name as the action symbol and takes no parameters.

The boilerplate for an action routine for action symbol *A* is:

```
procedure A( )
local V,e,...
V:=popSem(...)
if e:=anyError(V) then {pushSem(e); return}
...
pushSem(...)
return
end
```

The action routine is a parameterless procedure with the same name as the action symbol. It pops the appropriate number of values off the semantics stack. If there is an error token among them, then there was an error in a subphrase, so the action routine pushes an error token back on the stack and returns. Otherwise it performs whatever action it should and pushes a value back on the stack.

Of course, the pushSem's should be omitted if the action routine isn't supposed to leave any value on the stack.

Also, you might not call `anyError(V)` if you want to recover from some syntactic errors. For example, the parser's error recovery might insert a ")" before a ";". As long as the expression preceeding the ")" is okay, you might want your compiler to go ahead and generate code.

7.5 Interface to rptperr.icn

procedure reportParseError(t)

called by the parser

parameter: t—a token encountered by the parser that it wasn't expecting

returns nothing

Actually, this is such a small procedure, we usually just include a copy of it with our main program rather than compiling it separately.

7.6 Main procedure

We will need to provide a main program to initialize our compiler and call the parser. Do what we do: adapt one that already exists. Here is the main program from the TCLLk parser generator:

Figure 16 Example main program for a compiler.

```
 1 # TCLLk -- an LL(1) parser generator
 2 # Main program.
 3 # (written by Dr. Thomas W. Christopher)
 4 #
 5
 6 link readLLk,parseLLk,scangram,semgram,semstk,gramanal,LLk
 7
 8 procedure main(L)
 9 local filename,baseFilename,flags,filenameParts
10
11 flags := ""
12 if L[1][1]=="-" then {
13   flags := L[1]
14   filename := L[2]
15 } else {
16   filename:=L[1]
17 }
18 if /filename then
19   stop("usage: iconx tcLLk [flags] filename.grm")
20
21 filenameParts:=fileSuffix(filename)
22 baseFilename:=filenameParts[1]
23 if filename==(baseFilename||".LLk") then
24   stop("would write output over input")
25 initScanner( filename |
26   (/filenameParts[2] & baseFilename||".grm")) |
27     stop("unable to open input: ",filename)
28
29 initGrammar()
30 initSemanticsStack()
31
32 parseLLk(readLLk("tcLLk.LLk"))
33
34 finishDeclarations()
35 LLk(baseFilename||".LLk")
36 if find("p",flags) then printGrammar()
37 write(errorCount," error",(errorCount~=1&"s")|"",
38  " and ",warningCount," warning",(warningCount~=1&"s")|"")
39
40 end
41
42 # From:     filename.icn in Icon Program Library
43 # Author:   Robert J. Alexander, 5 Dec. 89
44 # Modified: Thomas Christopher, 12 Oct. 94
45
46 procedure fileSuffix(s,separator)
47    local i
48    /separator := "."
49    i := *s + 1
50    every i := find(separator,s)
51    return [s[1:i],s[(*s >= i) + 1:0] | &null]
52 end
```


Note:

3 Lines 11-19 read and check the input file name and optional flags.

4 Lines 21-24 decompose and check the input file name.

5 Lines 25-27 try to open the input file. Procedure initScanner will fail if the file can't be opened.

6 Line 29 initializes the semantics module, which contains the action routines.

7 Line 30 initializes the semantics stack in module semstk.icn.

8 Line 32 reads the TCLLk parse tables and calls the parser.

9 Lines 34-38 finish processing the user grammar.

10 Lines 42-52 are adapted from the Icon programming library to separate an extension from a base file name.

7.7 Structure of scanner

We must provide a parameterless procedure, *scan*, which will return the next token from the input each time it is called. We will probably wish to provide with it a procedure *initScanner* which will open the input file and initialize the scanner. We ourselves call that routine from the main program, so we can choose whatever interface we want for it.

As with main programs, we probably will not write an entirely new scanner when we need one; we will adapt one that already exists. Here is the scanner we usually start with, written for the language Oberon-2:

Figure 17 Example scanner.

```
 1 #
 2 # Scanner for Oberon 2
 3 #
 4
 5 global inputFile
 6 global inputLine,inputLineNumber,inputColumn,eoiToken
 7 global keywordSet
 8
 9 procedure initScanner(filename)
10 inputFile := open(filename,"r") |
11   stop("unable to open input: ",filename)
12 return
13 end
14
15 procedure fractionPart()
16 suspend "." || (tab(many(&digits)) | "")
17 end
18
19 procedure scaleFactor()
20 suspend tab(any('ED')) || (tab(any('+-')) | "") || tab(many(&digits))
21 end
22
23 procedure scan()
24 local t,c,b
25 static whiteSpace,initIdChars,idChars,hexdigits,commentDepth,commentLineNo
26 initial {
27   /inputFile := &input
```



```
28   inputLineNumber := 1
29   inputColumn := 1
30   inputLine := read(inputFile)
31   eoiToken := &null
32   whiteSpace := &ascii[1:34]#control ++ blank
33   initIdChars := &letters
34   hexdigits := &digits ++ 'ABCDEF'
35   idChars := &letters ++ &digits ++ '$_'
36   keywordSet := set([
37          "ARRAY","BEGIN","BY","CASE","CONST","DIV","DO",
38          "ELSE","ELSIF","END","EXIT","FOR","IF","IMPORT",
39          "IN","IS","LOOP","MOD","MODULE","NIL","OF","OR",
40          "POINTER","PROCEDURE","RECORD","REPEAT","RETURN",
41          "THEN","TO","TYPE","UNTIL","VAR","WHILE","WITH"
42  ])
43  }
44  if \eoiToken then return eoiToken
45  repeat inputLine ? {
46   tab(inputColumn)
47   tab(many(whiteSpace))
48   c := &pos
49   if b := tab(many(&digits)) then {
50          if b ||:= tab(many(hexdigits)) || ="X" then {
51                 t := Token("character",b,
52                        inputLineNumber,c)
53          } else if b ||:= tab(many(hexdigits)) || ="H" then {
54                 t := Token("hexinteger",b,
55                        inputLineNumber,c)
56          } else if b := b || fractionPart() ||
57              scaleFactor() then {
58                 t := Token("real",b,
59                        inputLineNumber,c)
60          } else if b ||:= fractionPart() then {
61                 t := Token("real",b,
62                        inputLineNumber,c)
63          } else if b ||:= "." || scaleFactor() then {
64                 t := Token("real",b,
65                        inputLineNumber,c)
66          } else {
67                 t := Token("integer",b,
68                        inputLineNumber,c)
69          }
70          inputColumn := &pos
71          return t
72   } else
73    if any(initIdChars) then {
74          t := Token("ident",tab(many(idChars)),
75                 inputLineNumber,c)
76          inputColumn := &pos
77          if member(keywordSet,t.body) then
78                 t.type := t.body
79          return t
80   } else
81    if b := =(":=" | ">=" | "<=" | "..") then {
82          inputColumn := &pos
83          return Token(b,b,inputLineNumber,c)
84   } else
85    if ="(*" then {
86          inputColumn := &pos
87          commentDepth := 1
88          commentLineNo := inputLineNumber
89          while commentDepth > 0 do {
90                 tab(upto('*(')|0)
91                 if pos(0) then {
```



```
 92                    &pos := 1
 93                    inputLineNumber +:= 1
 94                    if not (&subject :=
 95                              inputLine := read(inputFile))
 96                           then {
 97                       eoiToken := Token("EOI","EOI",
 98                              inputLineNumber,1)
 99                       write("end of input in comment beginning at ",
100                              commentLineNo)
101                       return eoiToken
102                    }
103                 } else if ="*)" then {
104                       commentDepth -:= 1
105                 } else if ="(*" then {
106                       commentDepth +:= 1
107                 } else {
108                       move(1)
109                 }
110          }
111          inputColumn := &pos
112    } else
113     if b := tab(any(',=#()[]{}~+-*/|&^;:><.')) then {
114          inputColumn := &pos
115          return Token(b,b,inputLineNumber,c)
116    } else
117     if pos(0) then {
118          inputColumn := 1
119          inputLineNumber +:= 1
120          if not (inputLine := read(inputFile)) then {
121                eoiToken := Token("EOI","EOI",
122                              inputLineNumber,1)
123
124                return eoiToken
125          }
126    } else
127     if ="\"" then {
128          b := tab(find("\""))
129          if not( = "\"" ) then {
130                write("unterminated string at ",
131                       inputLineNumber," ",c)
132          }
133          t := Token("string",b,inputLineNumber,c)
134          inputColumn := &pos
135          return t
136    } else
137     if ="'" then {
138          b := tab(find("'"))
139          if not( = "'" ) then {
140                write("unterminated string at ",
141                       inputLineNumber," ",c)
142          }
143          t := Token("string",b,inputLineNumber,c)
144          inputColumn := &pos
145          return t
146    } else
147          {
148          write("unexpected character: ",move(1),
149                 " at line ",inputLineNumber," column ",c)
150          inputColumn := &pos
151    }
152 }
153 end
```

Notes:

11 Lines 9-13 are the initialization routine, *initScanner*, that tries to open the input file.

12 Lines 15-21 help in recognizing real numbers.

13 Lines 23-153 are the scanner proper.

14 Lines 26-43 initialize the scanner the first time it is called. They could have been included in *initScanner* if the static's on line 25 had been made global.

15 Line 44 checks to see if an end-of-input token has been returned yet. If so, it returns it again. We don't keep trying to read past the end of file.

16 Line 45 is a repeat because when we fall off the end of an input line, we will have to read in a new line and restart our scan at its beginning. We make inputLine the subject string and enter the compound expression to look for tokens.

17 Line 46 moves the cursor &pos over to the next column to look in.

18 Line 47 moves the cursor past any white space.

19 Line 48 remembers where the first legible character was so that we can report it as the *column* in a Token record.

20 Lines 49 -151 are a nested if statement to find tokens. The token types are grouped by the class of character they begin with.

21 Lines 49-72 handle all tokens that begin with a digit.

22 Lines 50-53 handle characters written in hexadecimal format.

23 Lines 53-56 handle integers written in hexadecimal format.

24 Lines 56-60 handle real numbers with both a fraction part and an exponent.

25 Lines 60-63 handle real numbers with a fraction part but no exponent.

26 Lines 63-66 handle real numbers with an exponent but no fraction.

27 Lines 66-69 handle integers.

28 Line 70 remembers where to restart the scan on the next call.

29 Lines 73 through 80 handle identifiers and keywords. A keyword is simply an identifier that is found in the set keywordSet.

30 Lines 81-84 handle two character operators.

31 Lines 85-112 handle comments, which in Oberon-2 are delimited by (* and *) and can extend over multiple lines and be nested. Following the comment, this code falls out of the if expression to repeat the search for a token from the beginning.

32 Lines 113-116 handle single character operators and punctuation.

33 Lines 117-126 handle the scanner falling off the end of the line. (See also lines 91-103 which handle the same thing within a comment.)

34 Lines 127-146 handle quoted strings.

35 Lines 146-151 handle the default case of an unexpected character in the

input.

Appendix A The TCLLk input grammar

Here is a grammar for TCLLk's input:

```
start = grammar.
grammar =  { declaration }.
declaration = ID ( ":" rhs "." | "=" alts ".").
rhs = {elem}.
alts = rhs {"|" rhs}.
elem =  ID [ "!" ] | "(" alts ")" | "{" alts "}"  | "[" alts "]" .
```

In the grammar, *ID* represents an identifier or a quoted string of special characters (recognition of *ID*s is handled by the scanner). The syntax

```
declaration = ID ":" rhs "." .
```

is a form of declaration that gives the symbols on the right hand side of the ":" special meanings. There are four such declarations:

- start : ID .

  This declares the identifier *ID* to be the start symbol. It is equivalent to "*start = ID* ."

- EOI : ID .

  This declares symbol *ID* to represent end-of-input. If this is not provided, the parser generator declares *EOI* itself to be the end-of-input symbol.

- actions : ID1 ID2 ... IDn .

  This declares the identifiers to be action symbols so they can be used without following them with exclamation points.

- fiducials: ID1 ID2 ... IDn .

  This declares the identifiers to be *fiducial symbols* for use in panic mode error recovery. Error recovery was discussed in Chapter 6 on page 55 .

Identifiers can have two forms:

- A letter or underscore ("_"), followed by zero or more letters, digits, or underscores.

- A string of any characters except a quote enclosed in (double) quotes, e.g. "=".

An identifier must be entirely on one line.

A comment is the same as in Icon: a # and all the characters following it up to the end of the line.

Appendix B Contents of the LLk record

The best way to use TCLLk to generate a parser in some language other than Icon is to simply run the parser generator and write a program in Icon to read in the tables and translate them into the other language. To do that, you need to know the contents of the *LLk* record returned by procedure *readLLk*.

The record definition is:

```
record  LLk(sel,deflt,
     terminals,actions,
     fiducials,firstFiducials,
     minLengRHS,
     start,eoi)
```

All symbols are represented by character strings, their names. The fields are as follows

- *start* is the start symbol.
- *eoi* is the end-of-input symbol.
- *terminals* is a set containing all the terminal symbols.
- *actions* is a set containing all the action symbols.
- *sel* is a table used to select which right hand side to use for a nonterminal on the stack and a terminal in the input. Let *L* be the LLk record, *N* be the nonterminal, and *T* be the terminal, then if *L.sel[N]* is not *&null* and if *L.sel[N][T]* is not *&null*, then *L.sel[N][T]* is a list of symbols to replace *N* with—the right hand side. However, if either *L.sel[N]* is *&null* or *L.sel[N][T]* is *&null*, there may still be a replacement right hand side given by field *deflt*.
- *deflt* is a table to specify default right hand sides for nonterminals. It is used only if the -d flag was specified on the command line. Let *L* be the LLk record, *N* be the nonterminal, and *T* be the terminal. The parser will first try to look up a right hand side in *L.sel[N][T]*. If there is no right hand side there, the parser tries to find one in *L.deflt[N]*. If *L.deflt[N] is* not *&null*, the parser will replace *N* with the list of symbols in *L.deflt[N]*. The whole purpose of this table is to save space in the *sel* table. It is used under two circumstances: (1) for nonterminals that have only one production or (2) for the right hand side chosen by the largest number of terminal symbols.

- *fiducials* is a set containing all the fiducial symbols, i.e., the subset of terminal symbols at which the parser will try to resume parsing following an error.
- *firstFiducials* is a table mapping nonterminals into the sets of fiducial symbols they derive first. The error recovery uses this when it scans ahead to a fiducial and then sees if the fiducial is predicted. A fiducial is predicted if it is on the prediction stack or if a nonterminal is on the stack which can derive the fiducial first.
- *minLengRHS* is a table mapping each nonterminal to one of its right hand sides which will derive a minimum length terminal string. It is used by the error recovery to replacement tokens for the tokens thrown away during panic mode error recovery.

Care has been taken to minimize the storage required by the parsing tables. All occurrences of the same right hand side are represented by the same list (not merely lists with the same contents). All symbols are represented by the same bytes in Icon's string area, not merely by equal strings.

