Compiler Design in C

Allen I. Holub

Software Distribution 1.04.

Dear Folks,

The first thing is the software license. I've deliberately put this into English rather than legaleeze, and trust you to not take advantage of my desire to communicate clearly. A less restrictive license is available for the asking to academic institutions that use *Compiler Design in C* as a required textbook. Just drop me a note at Software Engineering (address below).

SOFTWARE LICENSE FOR COMPILER DESIGN IN C

Using the enclosed software constitutes agreement to the following terms and conditions.

(1) You are licensed to use the software on a single-user computer. You may not install it on a network or a multi-user computer system. You may copy the software as necessary for backup purposes. "Use it like a book" as they say.

(2) **None of the software on this disk may be used in the design or fabrication of weapons, weapon components, or weapon-delivery systems, or in any research that will contribute directly to the design or fabrication of weapons or weapon-delivery systems. This software may not be used by any company that produces or sells weapons, weapon components, or weapon-delivery systems, even if the software will not be used in weapons applications. The foregoing does not apply to a company that produces a product of general utility that happens to be used in a weapon manufactured by a different company.**

(3) This software is not in the public domain. (The author, Allen I. Holub, is the owner of the software and has the right to enter into this agreement.) You may not distribute it to anyone, except as follows:

a. **You must first pay for the code** by sending a check for $79.95 (plus local sales tax if you are in Califorina) to

    Allen Holub
    c/o Software Engineering Consultants
    P.O. Box 5679
    Berkeley, CA 94705

You may also pay by MasterCard or Visa, or use the on-line form found at *http://www.holub.com/compiler/compiler.html*.

b. You may distribute the output from LeX, llama, and occs freely, and in any form, including source and binary form. You may incorporate this output into any of your programs and distribute those programs freely in source or binary form.

c. You may incorporate any of the source code into your own programs, and distribute binary versions of those programs without restriction. You may not, however, distribute the source-code itself, and the programs that you do distribute must be substantially different from LeX, llama, and occs. (For example, you may not use the sources to create a compiler-compiler or a lexical-analyzer generator which you then distribute.) This restriction applies both the source code as distributed, and to any versions of the source code that you create by modifying the distributed version.

d. The documentation for all products that you distribute that incorporate any of the code on the distribution disks, in original or modified form, must state that the product uses code from *Compiler Design in C* by Allen I. Holub and that the incorporated code is © 1990, by Allen I. Holub. You (the customer) retain copyright to your own original work, of course.

e. You may distribute bug fixes freely, but only if you also report them to me directly (see address, below), You may distribute only enough of the code to identify where the bug is and how to fix it—a UNIX-style “diff” file is ideal. I’d prefer for you to send internet postings directly to me. I’ll digest them and post them to the net at periodic intervals.

(4) LeX, occs, and LLama, are trademarks of Allen I. Holub and must be identified as such if used by you.

(5) The author (Allen Holub) has done his best to assure that the programs on the disk are as error free as possible. Nonetheless, the author makes no warranty of any kind, expressed or implied, with regard to these programs or the documentation. The author shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs. In plain words, if a program on this disk goes crazy and does something like destroy every file on your hard disk, it’s your fault for not backing up properly. Don’t blame the program.

INSTALLATION

The enclosed disks contain all of the source code from *Compiler Design in C* and executable versions of the tools (LeX, LLama, and occs). There is also an executable visible-parser version of the compiler in Chapter 6. All this material has been compressed in order to save disks when shipping. The unpacking process creates many files and several subdirectories (which can be rooted anywhere in your file system). The software is shipped on either two 360K 5¼" disks or one 720K 3½" disk. Installation is performed automatically using the following procedure:

(1) Put the installation disk into a floppy drive, the A: drive is the default one, but another can be used.

(2) Create the directory in which you want to install the Compiler-Design file system and go there. If, for example, you want to put everything in and under ’c:/compiler’, issue the following commands to DOS:

```
c:
mkdir \compiler
cd \compiler
```

(3) Copy the file `install.bat` from the distribution disk to the target directory:

```
copy a:install.bat
```

(4) Execute the installation batch file. If the distribution disk is in the A: drive, just type:

```
install
```

If the installation disk is in another drive, type

```
install D:
```

where *D:* is the drive that holds the installation disk. If you’re installing from 5¼" disks, change disks when prompted.

The distribution disk expands to about a megabyte of stuff. If you’re installing from 5¼" disks, you’ll need another megabyte or so of scratch space to do the installation, but that space is freed up once the installation is complete.

ABOUT THE DISTRIBUTION DISK

The installation process creates several directories (the ... represents the directory used as the root of the file system):

`.../bin`	Various executable programs (discussed below)
`.../include`	Various system-level **`#include`** files
`.../include/tools`	Various compiler-related **`#include`** files
`.../lib`	parser templates for LeX, llama, and occs. You should also move the libraries discussed below to this directory once you have compiled them.
`.../src`	Sources for the cc program, discussed below
`.../src/compiler`	
`.../src/compiler/c`	Sources for the C compiler in Chapter Six.
`.../src/compiler/lex`	Sources for LeX, from Chapter Two.
`.../src/compiler/lib`	Library subroutines from most chapters and Appendix A.
`.../src/compiler/parser`	Sources for llama and occs from Chapters Five and Six
`.../src/tools/curses`	Sources for the curses window-management library.
`.../src/tools/termlib`	Sources for low-level I/O functions used by curses.
`.../work`	Miscellaneous files that didn't fit anywhere else.

The various executables found in *.../bin* are:

`occs.exe`	An executable version of occs
`llama.exe`	An executable version of llama
`lex.exe`	An executable version of lex
`cc.exe`	This is a driver program that corrects some of the command-line deficiencies in the Microsoft cl program. It compensates for the fact that Microsoft's cl ignores the current SWITCHAR setting, and is used to invoke cl from the makefiles. **cc** translates into backslashes all forward slashes that appear between the "cc" and an optional "-link" and then chains to cl. For example: `cc /src/compiler/c/foo.c -link /NOE tools.lib` effectively does the following: `cl \src\compiler\c\foo.c /link /NOE tools.lib`
`llexpr.exe`	A "visible" parser for the llama example in Appendix E. Create a test file containing a semicolon-terminated arithmetic expression made up of numbers, names, parentheses, and the + and * operators; like this: `1+(2+3*4);` (don't forget the semicolon). Execute the parser by saying `llexpr test`, where `test` is the name of your test file.

`yyexpr.exe`	A "visible" parser for the occs example in Appendix E. Use it just like llexpr, but don't put the semicolon at the end of the expression.
`c.exe`	An executable "visible" parser for the C compiler in Chapter 6. You can use it both to see how the IDE works and to follow along with the examples in Chapter Six without having to create the C compiler from scratch.
`pp`	This is a UNIX C-shell script that simulates an ANSI preprocessor by using cpp and sed. It is discussed further, below.

If you just intend to use the tools, without actually recompiling them, you can create a minimal system with the following:

`.../bin/lex.exe` `.../bin/llama.exe` `.../bin/occs.exe`	Executable tools
`.../lib/*.par`	Template files. You should either initialize the LIB environment (a semicolon-delimited list of directory names similar to the PATH) to contain the name of this directory or move the *.par* files to some directory already listed in your LIB environment.
`.../lib/l.lib`	Must be linked to programs that have occs-generated parsers or LeX-generated lexical analyzers. You have to create this library from the provided sources—the process is discussed below.
`.../lib/curses.lib` `.../lib/termlib.lib`	These libraries must be linked to all occs/llama generated parsers that were compiled with YYDEBUG set (or which had a -D specified on the occs/llama command line). This is the windows system used by the IDE. You have to create these libraries from the provided sources—the process is discussed below.

COMPILING

All sources on the disk compile without modification under both Microsoft C, versions 5.1 and 7.0, Borland C++, ver. 2.0 (using only the C-compiler component of the system), and the UNIX BSD 4.3 compiler. It should also compile under Microsoft 6.0 and Borland 3.0, but I haven't actually done so, so can't promise anything. Similarly, the output from LeX, LLama, and occs compiles in all these environments. If you have any problems compiling with these compilers, please contact me so I can fix the source disk in future software versions.

If your compiler isn't in the foregoing list, but is ANSI compatible, then your best bet is probably to pretend you're the Microsoft compiler by adding the following lines at the top of *debug.h:*

```
#define MSDOS 1
#define _MSC_VER 700
```

Some massaging might still be necessary, however.

The compilation environment is selected by means of various macros in *debug.h*. The compiler assumes Borland C if `__TURBOC__` is defined. It assumes Microsoft ver. 5.x if `MSDOS` is defined and MSC_VER is not defined, Microsoft ver. 6.x if `MSDOS` is defined and `MSC_VER` is defined as 600, Microsoft ver. 7.x if `_MSDOS` is defined and `_MSC_VER` defined as 700. UNIX (ie. K&R C) is assumed if none of the above are present. `__TURBOC__`, `MSDOS` or `_MSDOS`, and `MSC_VER` are all defined automatically by the Borland or Microsoft compilers—you don't have to define them yourself. Search the sources for instances of the macros in Table 0.1 to find implementation-dependent code:

Table 0.1. Conditional-Compilation Macros

	Selects	**Description**
Compiler	`MSC(x)`	Argument is expanded if any Microsoft C compile
	`MSC5(x)`	Argument is expanded only if Microsoft C ver. 5 is active.
	`MSC6(x)`	Argument is expanded only if Microsoft C ver. 6 is active.
r is active.	`MSC7(x)`	Argument is expanded only if Microsoft C ver. 7 is active.
	`BCC(x)`	Argument is expanded only any Borland/Turbo C/C++ is active.
	`MSDOS`	Defined only if Microsoft C is active.
	`__TURBOC__`	Defined only if Borland C++ is active.
Operating System	`UNIX(x)`	Argument is expanded in UNIX environment.
	`MS(x)`	Argument is expanded in MS-DOS environment.
Language	ANSI(x)	Argument is expanded only if an ANSI compiler is active.
	`KnR(x)`	Argument is expanded only if a K&R C compiler is active.
`ANSI()` and `KnR()` are mutually exclusive (they both won't be defined at the same time). Same goes for `MS()` and `UNIX()`, `MSC()` and `BCC()`, etc. `MSDOS` and `__TURBOC__` are generated automatically by the Microsoft and Borland compilers.		

The executables on the disk were compile under Microsoft 7.0.

I am assuming that you'll use some form of make to do your compilation. Three versions of the makefiles appear in the various source-code directories. The file called *makefile* is for Microsoft C and Microsoft nmake. The *borland.mak* file is for Borland C++ and the version of make that's shipped with that compiler. Finally, *unixmake.mak* is the UNIX makefile.

Note that the makefile in *.../src/compiler/c* makes a debugging version of the c compiler. To make a non-debugging version, you'll have to remove the -D from all the occs invocations in the makefile. You'll also need to stop the linker from calling in *yydebug.obj* by modifying *.../src/compiler/c/main.c* as follows: comment out the `yy_get_args()` call in `main()` and also remove the `yyhook_a()` and `yyhook_b()` subroutines.

MAKING THE CODE USEABLE

To use LeX, occs, and LLama, you'll need to compile the run-time libraries used by the compiler-compiler's output code. The sources for the libraries are in the following directories:

Library:	**Sources are in:**
curses.lib	*. . . /src/tools/curses*
termlib.lib	*. . . /src/tools/termlib*
l.lib	
comp.lib	*. . . /src/compiler/lib*

Makefiles for Microsoft's nmake and Microsoft C are provided in the various directories—as are makefiles for Borland C++ and UNIX. You don't have to use them, however. To create *curses.lib* just compile everything in *. . ./src/tools/curses* and put the objects together in a library. You'll probably want to specify `-DA` on the compiler's command line to create an autoselect version of the library. (Most compilers create an implicit **`#define`** `A` when given `-DA`.) Create *termlib.lib* in a similar way, just compile everything in *. . ./src/tools/termlib* and put the objects together in a library. No special command-line switches are required here. The final two libraries can be combined to make your life easier. Instead of creating both *l.lib* and *comp.lib*, just compile all the sources in *. . ./src/compiler/lib* and put the objects into a combined library called something like *lcomp.lib*. Specify *lcomp.lib* everywhere that either *comp.lib* or *l.lib* is required.

All the *.h* files in the book should be on the distribution disk in the *. . ./include* or *. . ./include/tools* directories. Most of the files in *. . ./include* are bogus ANSI include files included here for UNIX compatibility (stdarg.h, stdlib.h, io.h, process.h, etc.). These pseudo-ANSI files will deliberately cause error messages to be printed if an MS-DOS compiler tries to use them. Just delete everything in *. . ./include* except *curses.h* if you have problems.

All files included with quotes instead of brackets should be in the same directory as the file that includes them. If your compiler can't find an include file that uses <>, it's probably one that comes with Microsoft C or Borland C and you will have to figure out how to keep your own compiler happy. With the exception of *curses.h*, all of the nonlocal include files are in the */tools* subdirectory of *. . ./include* and are included in the programs with something like **`#include`** `<tools/debug.h>`. This means that you should set up your compiler so that it will search for include files in ". . ./include." The compiler will take care of adding the "/tools" when necessary. You can do this with the Microsoft C's `INCLUDE` environment like this:

```
set LIB=d:\lib;d:\c700\lib
set INCLUDE=d:\include;d:\c700\include
```

The first directory in each environment tells the Microsoft where to find the compiler-system's files, the second tells the compiler where to find its own files. Do the same thing with Borland C++'s *turboc.cfg* file. My turboc.cfg looks like this:

```
-ID:\BORLANDC\INCLUDE
-LD:\BORLANDC\LIB
-ID:\INCLUDE
-LD:\LIB
-w-eff
-w-pia
-wpro
-wnod
-wstv
-wuse
-w-inl
-wpre
```

The first two lines tell the compiler where to find its own include files and libraries. The second two lines tell it where to find the compiler-system's include files and libraries.

The remainder of the file disables two annoying warning messages and enables a bunch of useful, but normally disabled messages.

Note that the LeX, llama, and occs output files contain **`#include`**s for the *<tools/l.h>* and *<tools/compiler.h>* files that are on the disk. You must either make sure that your compiler can find these files, or modify the *.../lib/`*.par` files so that the includes aren't necessary. (The latter is probably the best approach for* UNIX *systems.) l.h* is included only for prototypes for the `ii_` and `yy` subroutines in the library, so can be discarded in a UNIX system without difficulty. *debug.h* is needed to include *l.h*, but the `KnR()`, `ANSI()`, `P()`, and `BCC()` macros from *debug.h* are also used in the *.par* files.

UNIX USERS:

First, you can dispense with all of the curses and termlib stuff on the distribution disk. Just use the system's *−lcurses* and *−ltermlib* libraries when you're linking.

The code compiles and runs without difficulty under the BSD 4.3 C compiler. Just don't create a **`#define`** for `MSDOS` or `__TURBOC__` anywhere. I can't vouch for other UNIX versions and other compilers. The code in *yydebug.c* has been tested under Berkeley curses, not System V curses, but so far, this has not caused problems. The *system.v* file in the *.../src* directory of the unpacked file system describes one readers change's to the code to get it to compiler under System V.

It easiest to rename *unixmake.mak* to *makefile* before attempting a compile using make.

The various makefiles assume that the compiler sources are rooted at */violet_b/holub/compiler*. When I say something like *. . ./bin*, I mean */violet_b/holub/compiler/bin*, and so on. You will have to modify the makefiles to reflect your own root directory. I do not recommend moving the source-code files to directories other than the default ones unless you want to do a major makefile *peristroika.*

There are a few things that you have to deal with to port the code. First, the UNIX compiler can not handle ANSI-style token pasting. A shell script called *. . ./bin/pp* takes care of this problem. It uses a combination of the preprocessor and **sed** to simulate token pasting. Use it like this:

```
pp source_file_name cc_command_line_switches
```

It generates a pre-preprocessed file that can then be submitted to cc. (The generated file's name is made up of the root part of the original file name with a `.pp.c` extension added). *pp* generates **`#line`**s to make error messages from cc reference the source file, not pp's output file. There's an example of how *pp* is used in *. . ./src/compiler/parser/unixmake.mak*.

The other problem is the lama.par and occs.par files in *. . ./lib*. They both include <tools/yystack.h>, which uses token pasting. Its clearly unacceptable to require a user of LLama or occs to run all of the output through pp, however. For this reason, alternate versions of these two files (called *occspar.unx* and *llamapar.unx*) are also provided in *. . ./bin* on the distribution disk. If your compiler doesn't support ANSI token pasting, you should replace the *.par* files with the alternate ones. The *.unx* versions of the file are rather weird looking because they've been sent through a preprocessor. (There are no comments, lots of blank lines, and a few very long lines that represent macro expansions). You might want to save the original versions somewhere for documentation purposes.

Finally, don't forget to initialize the LIB environment to hold the directory name for the directory where the *.par* files are found. Do this in the C shell with:

```
set LIB pathname
```

Do it in the Bourne and Korn shells with:

```
LIB=pathname
export LIB
```

There's one known bug in the UNIX version of yydebug.c, which supports the occs/llama interactive debugging environment. Occasionally, when you're parsing madly away after executing a `g` command, and you hit a key to stop the parse, a newline that should go to the "comments" window is lost, and a weird-looking line is created. I've been getting nowhere trying to fix the problem, and suspect a bug in curses. If any of you curses gurus out there can suggest a solution, I'd appreciate it.

OTHER STUFF

You may have problems using `stdin` with a LeX-generated scanner. The `ii_fillbuf()` routine in */src/compiler/lib/input.c* was assuming that it had reached end of file if the number of characters received from `read()` wasn't the same as the number of characters requested. Under MS-DOS a `read()` from standard input almost always returns a value smaller than the requested number of bytes, however. I've fixed the problem in the MS-DOS environment by replacing the line

```
if( got < need )
```

[towards the bottom of `ii_fillbuf()`] with:

```
if( got < need MS( && eof(Inp_file) ))
```

The `eof()` function returns true if the indicated stream (not `FILE`) is at end of file. There is no UNIX equivalent and this fix is not particularly portable.

BUGS, UPDATES, COMMUNICATION

At this writing, the tools (LeX, occs, llama, their output, and the libraries used by them) are all error free, as far as I know. There are a few bugs in the compiler presented in Chapter 6 that will be fixed in a future release (See Exercise 6.2 on page 652). Note that the grammar (and the compiler, by extension), needs to be modified a bit to bring it into full ANSI compliance. I'll fix this problem in a second edition of the book, but have decided to leave well enough alone for now so that the enclosed compiler will behave as described in Chapter 6. I didn't want the compiler that you're reading about to parse differently than the distributed compiler.

Please report any bugs or compilation problems to me, Allen Holub c/o Software Engineering, P.O.Box 5679, Berkeley, CA 94705, or via internet (holub@violet.berkeley.edu). Compuserve users can access internet from the email system by using:

```
>INTERNET:holub@violet.berkeley.edu
```

as my address. (Type `help internet` for more information.) I'll fix any bugs as soon as possible and send you an update. Updates will also be available at regular intervals for people who don't have access to Internet or Compuserve. I'll send out postcards when appropriate.

– Allen Holub

Errata: ***Compiler Design in C***

This document is a list of typos and corrections that need to be made to *Compiler Design in C,* **Allen Holub, 1990 (as of September 11, 1997). The corrections marked "Disk only" represent changes made to the files on the distribution disk that either don't affect the code in any significant way or are too big to insert into the book. All these will eventually be incorporated into the second edition, if there is one, but they won't be put into subsequent printings of the first edition. There are also a many trivial changes that have been made to the code on the disk to make it more robust: Function prototypes have been added here and there, as have includes for .h files that contain prototypes, all .h files have been bracketed with statements like:**

```
#ifndef __FILE_EXT  /* file name with _ for dot */
#define __FILE_EXT
   ...
#endif
```

to make multiple inclusions harmless, and so forth. These changes allow the code to compile under Borland C++ as well as Microsoft C and BSD UNIX. None of these trivial changes are documented here.

The printings in which the error occurs and the software version in which the change was made are identified in the margin at the head of each entry. For example, the numbers next to the current paragraph indicate that the typo is found in both the first and second printings, and that the bug was fixed in software version 1.02. Determine your printing by looking at the back of the flyleaf, where you'll find a list of numbers that looks something like this: `10 9 8 7 6 5 4 3`**. The smallest number in the list is the printing.**

1, 2 (1.02)

Page xvi - *Seventh line from the bottom. Change "No credit cards" to "No purchase orders or credit cards." The last two lines of the paragraph should read:*

must add local sales tax. **No purchase orders or credit cards** (sorry). A Macintosh version will be available eventually. Binary site licenses are available for educational institutions.

Page xvi - *Last line. Internet can now be accessed from CompuServe. Add the Compuserve/internet address >INTERNET:holub@violet.berkeley.edu to the parenthesized list at the end of the paragraph. A replacement paragraph follows:*

The code in this book is bound to have a few bugs in it, though I've done my best to test it as thoroughly as possible. The version distributed on disk will always be the most recent. If you find a bug, please report it to me, either at the above address or electronically. My internet address is *holub@violet.berkeley.edu* CompuServe users can access internet from the email system by prefixing this address with *>INTERNET:* —type `help internet` for information. My UUCP address is *...!ucbvax!violet!holub*.

Page xvi - *15 lines from the bottom. Change the phone number to:*

(510) 540-7954

Page xviii - *Line 7, change "that" to "than". The replacement line follows:*

primitives. It is much more useful than **pic** in that you have both a WYSIWYG capability ‘
‘

Page 8 - *The line that starts "***JANE** *verb object" in the display at the bottom of the page is repeated. Delete the first one. A replacement display follows:*

sentence	apply *sentence*$\rightarrow$*subject predicate* to get:
subject predicate	apply *subject*$\rightarrow$*noun* to get:
noun predicate	apply *noun*$\rightarrow$**JANE** to get:
JANE *predicate*	apply *predicate*$\rightarrow$*verb object* to get:
JANE *verb object*	apply *verb*$\rightarrow$**SEES** to get:
JANE SEES *object*	apply *object*$\rightarrow$**noun op_participle** to get:
JANE SEES *noun opt_participle*	apply *noun*$\rightarrow$**SPOT** to get:
JANE SEES SPOT *opt_participle*	apply *opt_participle*$\rightarrow$*participle* to get:
JANE SEES SPOT *participle*	apply *participle*$\rightarrow$**RUN** to get:
JANE SEES SPOT RUN	done—there are no more nonterminals to replace

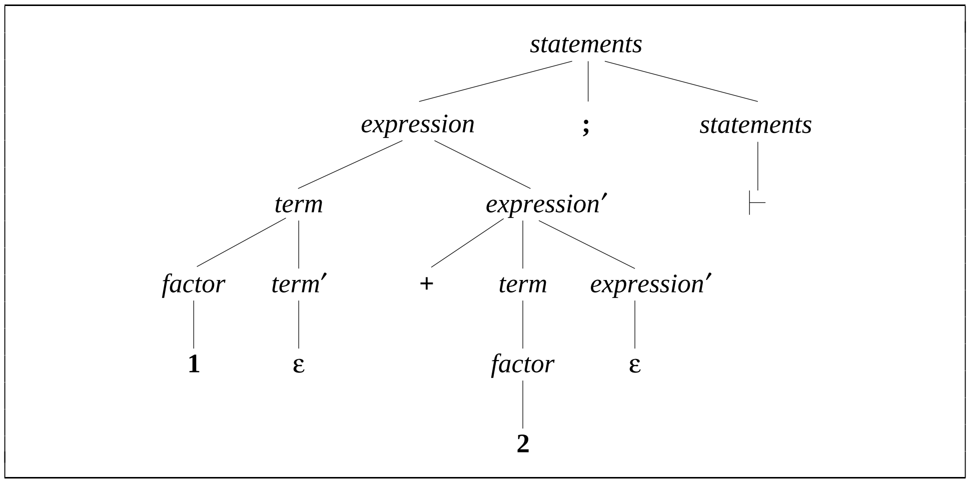
Page 11 - *Table 1.1, line 1 should read:* statements$\rightarrow$**EOF**. *A replacement table follows:*

Table 1.1. A Simple Expression Grammar

1.	*statements*	$\rightarrow$	**EOF**
2.		|	*expression* **;** *statements*
3.	*expression*	$\rightarrow$	*expression* **+** *term*
4.		|	*term*
5.	*term*	$\rightarrow$	*term* $*$ *factor*
6.		|	*factor*
7.	*factor*	$\rightarrow$	**number**
8.		|	**(** *expression* **)**

Page 12 - *Figure 1.6. Replace the figure with the following one:*

Figure 1.6. A Parse of `1+2`



Page 16 - *Listing 1.2, delete line 46. (Replace it with a blank line.)*

Page 18 - *Replace the untitled table just under Listing 1.4 with the following (only the first line has been changed):*

1.	*statements*	→	*expression* **;** *eoi*
2.		|	*expression* **;** *statement*
3.	*expression*	→	*term expression′*
4.	*expression′*	→	+ *term expression′*
5.		|	ε
6.	*term*	→	*factor term′*
7.	*term′*	→	∗ *factor term′*
8.		|	ε
9.	*factor*	→	**num_or_id**
10.		|	*(expression)*

Page 24 - *Line 110 of Listing 1.7 should read* `*p++ = tok;`

```
110             *p++ = tok;
```

Page 26 - *Change the caption to Figure 1.11 as follows:*

Figure 1.11. A Subroutine Trace of `1+2*3+4` (Improved Parser)

Page 27 - *Change Line 42 of Listing 27 to* `tempvar2 = term();`

```
42      tempvar2 = term();
```

Page 27 - *Replace the word* `temporary` in the code part of lines 19, 26, and 42 of Listing 1.9 with the word `tempvar`. These three lines should now read:

```
19      tempvar = expression()
26      freename( tempvar );
```

Page 36 - *Replace Lines 16 and 17 of Listing 2.1 with the following (I've added a few parentheses):*

```
16  #define get(stream) (Pbackp < &Pbackbuf[SIZE] ? *Pbackp++ : getc(stream)    )
17  #define unget(c)    (Pbackp <= Pbackbuf       ? -1        : (*--Pbackp=(c)) )
```

Page 44 - *Replace line 186 of Listing 2.5 with the following line:*

```
186     Next = sMark = eMark = END - 1;  paMark = NULL;
```

Page 55 - *Eleven lines from bottom. The display that says* `([ˆa-z]|\en)` should read as follows:

```
([ˆa-z]|\n)
```

Page 41 - *Modify line 113 of Listing 2.3 to the following:*

```
113     eMark    = END;  pMark = NULL;
```

Page 46 - *Replace lines 299 and 300 of Listing 2.6 with the following lines:*

```
        int need,   /* Number of bytes required from input. */
            got;    /* Number of bytes actually read.   */
```

Page 57 - *Line six should read as follows:*
causes a transition to State 1; from State 1, an `e` gets the machine to State 2, and an `i` ‘
‘

Page 57 - *Eleventh line from the bottom should read as follows:*

```
next_state = Transition_table[ current_state ][ input_character ];
```

Page 57 - *Last line of second paragraph should read as follows (delete the "s"):*
r, or *e* from State 0) are all implied transitions to a special implicit *error state*. ‘
‘

Page 63 - *Listing 2.11, lines 2 and 3: remove the semicolons.*

Page 68 - *The first line beneath the Figure should read as follows (the Figure and Listing numbers are wrong):*

table is shown in Figure 2.6 and in Listing 2.15. The `Yy_cmap[]` array is indexed by ‘
‘

Page 72 - *The first display (second and third lines) should read as follows:*

```
['0',2]  ['1',2]  ['2',2]  ['3',2]  ['4',2]
['5',2]  ['6',2]  ['7',2]  ['8',2]  ['9',2]  ['e',5]
```

Page 73 - *The first line of the third paragraph calls out the wrong line number. It should read as follows:*

The `YYERROR()` macro on line 151 of Listing 2.19 prints internal error messages. ‘
‘

Page 73 - *First paragraph, lines two and four. Both references to Listing 2.18 should be to Listing 2.19. A replacement first paragraph follows:*
The remainder of *lexyy.c* file is the actual state-machine driver, shown in Listing 2.19 The first and last part of this listing are the second and third parts of the Ctrl-L-delimited template file discussed earlier. The case statements in the middle (on lines 287 to 295 of Listing 2.19) correspond to the original code attached to the regular expressions in the input file and are generated by LeX itself.

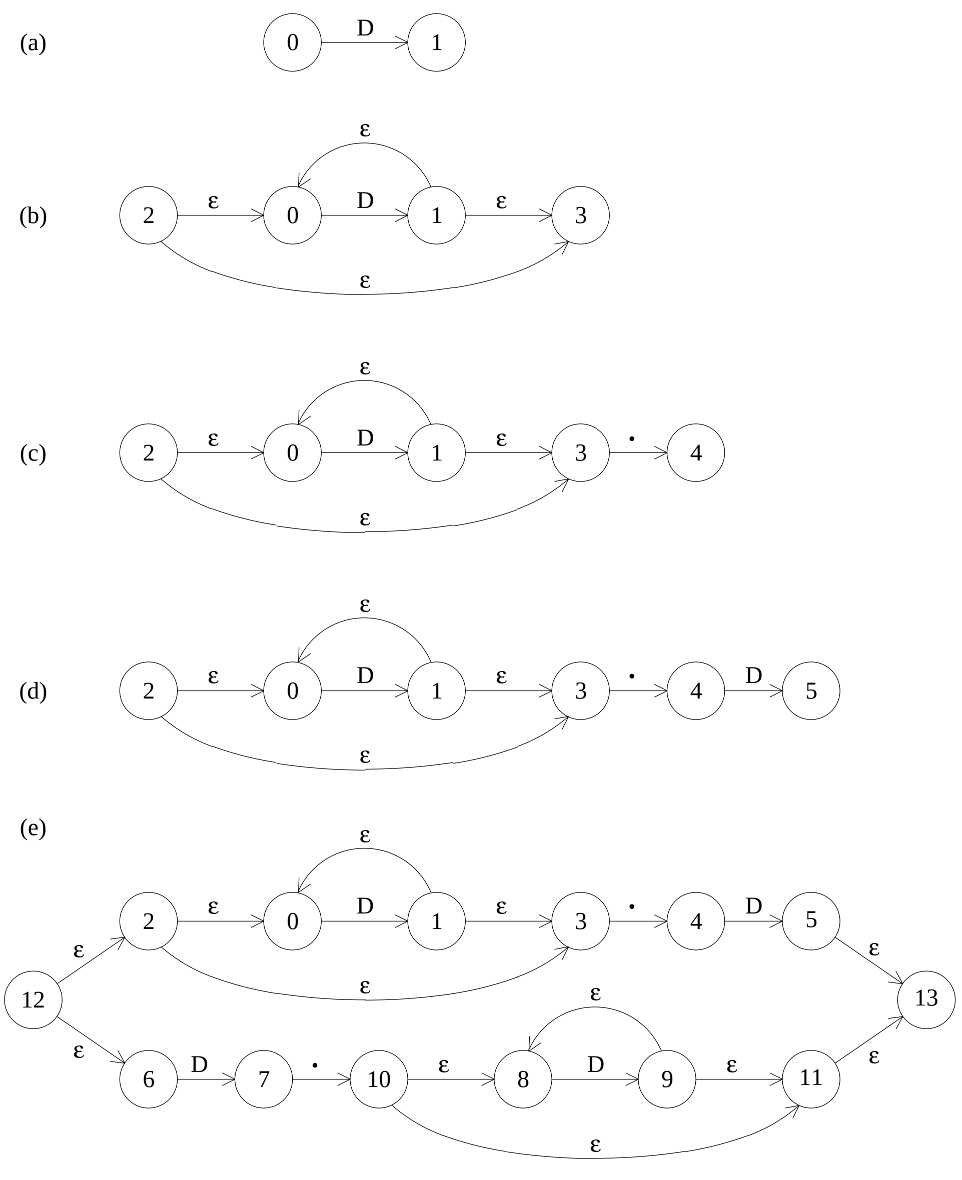
Page 76 - *Listing 2.19, line 166. Some code is missing. Replace line 166 with the following line:*

```
166       if( (c = ii_input()) && (c != -1) )
```

Page 78 - *Listing 2.19, lines 292 and 294. Align the* `F` in `FCON` under the `I` in `ICON` (on line 288).

Page 84 - *Figure 2.13(e). The arrows should point from* states 5 and 11 *to* state 13. Here is a replacement figure:

Figure 2.13. Constructing an NFA for (D∗\ .D|D\ .D∗)



Page 85 - *Third line from the bottom (which starts "For example") should read:*

For example, in a machine with a 16-bit `int`, the first two bytes of the string are the ‘ ‘

Page 86 - *Listing 2.21. Change the number* `512` on line 28 to `768`.

```
768
```

Page 91 - *Change the definition of STACK_USED on line 93 of Listing 2.25 to* `((int)(Sp-Sstack)+1)`. A replacement line follows:

```
 93  #define STACK_USED()  ( (int)(Sp-Sstack) + 1    )  /* slots used        */
```

Page 92 - *Change line 167 of Listing 2.25 to the following:*

```
167        if( textp >= (char *)strings + (STR_MAX-1) )
```

Page 95 - *Listing 2.26. Replace lines 280 to 283 with the following lines:*

```
280        *p = '\0';              /* Overwrite close brace. { */
281        if( !(mac = (MACRO *) findsym( Macros, *namep )) )
282            parse_err( E_NOMAC );
283        *p++   = '}';              /* Put the brace back.    */
```

Page 104 - *Second paragraph, first four lines (above the first picture) should read:*

Subroutines `expr( )` and `cat_expr( )` are in Listing 2.30. These routines handle the binary operations: | (OR) and concatenation. I'll show how `expr( )` works by watching it process the expression *A*|*B*. The `cat_expr( )` call on line 621 creates a machine that recognizes the *A:*

Page 121 - *Listing 2.38. Replace lines 218 to 255 with the following:*

```
    start_dfastate = newset();              /* 2 */
    ADD( start_dfastate, sstate );
    if( !e_closure( start_dfastate, &accept, &anchor) )
    {
    fprintf(stderr, "Internal error: State machine is empty\n");
    exit(1);
    }
    current = newset();                 /* 3 */
    ASSIGN( current, start_dfastate );

    /* Now interpret the NFA: The next state is the set of all NFA states that
     * can be reached after we've made a transition on the current input
     * character from any of the NFA states in the current state. The current
     * input line is printed every time an accept state is encountered.
     * The machine is reset to the initial state when a failure transition is
     * encountered.
     */

    while( c = nextchar() )
    {
    next = e_closure( move(current, c), &accept, &anchor );
    if( accept )
    {
        printbuf();             /* accept   */
        if( next );
        delset( next );         /* reset    */
        ASSIGN( current, start_dfastate );
    }
    else
    {                   /* keep looking */
        delset( current );
        current = next;
    }
    }
    delset( current    );   /* Not required for main, but you'll */
    delset( start_dfastate );   /* need it when adapting main() to a */
}               /* subroutine.              */
#endif
```

Page 122 - *First display, which starts with 'ε-closure({12})'', should read as follows:*

ε-closure({0}) = {0, 1, 3, 4, 5, 12} (new DFA State 0)

Page 123 - *Display on lines 6–9. Second line of display, which now reads 'ε-closure({7,11}) = {9, 11, 13, 14}," is wrong. The entire display should read as follows:*

DFA State 7 = {11}
ε-closure({11}) *= {9, 11, 13, 14}*
move({9, 11, 13, 14}, **.***)* *= ∅*
move({9, 11, 13, 14}, **D***)* *= {11} (existing DFA State 7)*

Page 124 - *First line of second paragraph should read:*

The maximum number of DFA states is defined on line seven of Listing 39 to be '
'

Page 129 - *First line should say "Listing 2.41", not "Listing 2.40" Second line should say "line 119", not "line 23". The forth line should say "line 129", not "line 31". A replacement paragraph follows:*

Several support functions are needed to do the work, all in Listing 2.41 The `add_to_dstates()` function on line 119 adds a new DFA state to the `Dstates` array and increments the number-of-states counter, `Nstates`. It returns the state number (the index in `Dstates`) of the newly added state. `in_dstates()` on line 139 of Listing 2.41 is passed a set of NFA states and returns the state number of an existing state that uses the same set, or –1 if there is no such state.

Page 129 - *Replace lines 147 and 148 of Listing 2.41 with the following:*

```
147       DFA_STATE *end = &Dstates[Nstates];
148       for( p = Dstates ; p < end ; ++p )
```

Page 130 - *Replace lines 193 and 194 of Listing 2.41 with the following*

```
193       DFA_STATE *end = &Dstates[Nstates];
194       for( p = Dstates; p < end ; ++p )
```

Page 139 - *Replace line 198 of Listing 2.44 with the following:*

```
198       SET    **end   = &Groups[Numgroups];
```

Replace Line 204 of Listing 139 with the following:

```
204       for( current = Groups; current < end; ++current )
```

Page 140 - *Replace lines 229 and 230 of Listing 2.45 with the following*

```
229       SET **end = &Groups[Numgroups];
230       for( current = Groups; current < end; ++current )
```

Page 152 - *Remove the* `#include "date.h"` on line 3.

Page 157 - *Replace line 117 with the following (I've added the* `--argc` on the left):

```
117     for( ++argv, --argc; argc && *(p = *argv) == '-'; ++argv, --argc )
```

Page 167 - *13th line from the bottom, "returns" should be "return."*

Page 167 - *Third line from the bottom. Change the second comma to a semicolon.*

Page 171 - *Replace the first display (which now reads noun*→**time**|**banana**) *with the following:*

noun → **fruit** | **banana**

Page 173 - *Second display, text to the right of the =L> is missing. It should read as follows:*

compound_stmt	=L>	**LEFT_CURLY** *stmt* **RIGHT_CURLY**
	=L>	**LEFT_CURLY RIGHT_CURLY**

Page 173 - *First line after second display, change expr*→ε *to stmt*→ε*: The line should read:*
The application of *stmt*→ε effectively removes the nonterminal from the derivation by ' '

Page 173 - *11th and 16th line from the bottom. Change* **CLOSE_CURLY** *to* **RIGHT_CURLY**
RIGHT_CURLY **RIGHT_CURLY**

Page 175 - *Figure 3.2. The line that reads '4*→**DIGIT** *error" should read '4*→**DIGIT** *5."*

5

Page 177 - *Listing 3.2, Line 12, should read:*

```
12      process_stmt( remember );
```

Page 178 - *Third line below the one that starts with "Since" (next to the marginal note), replace "the the" with "the": All three lines should read:*

Since a *decl_list* can go to ε, the list could be empty. Parse trees for left and right recursive lists of this type are shown in Figure 3.5. Notice here, that the *decl_list*→ε

Productions executed first or last.

production is the first list-element that's processed in the left-recursive list, and it's the last ‘ ‘

Page 179 - *fourth line of second display, "declarator" should be "declarator_list". Line should read:*

$$\textit{declarator\_list} \quad \rightarrow \quad \textbf{TYPE} \quad \textit{declarator\_list}$$

Page 180 - *Table 3.2. Grammatical rules for "Separator Between List Elements, Zero elements okay" row are wrong. Replace the table with the following one:*

Table 3.2. List Grammars

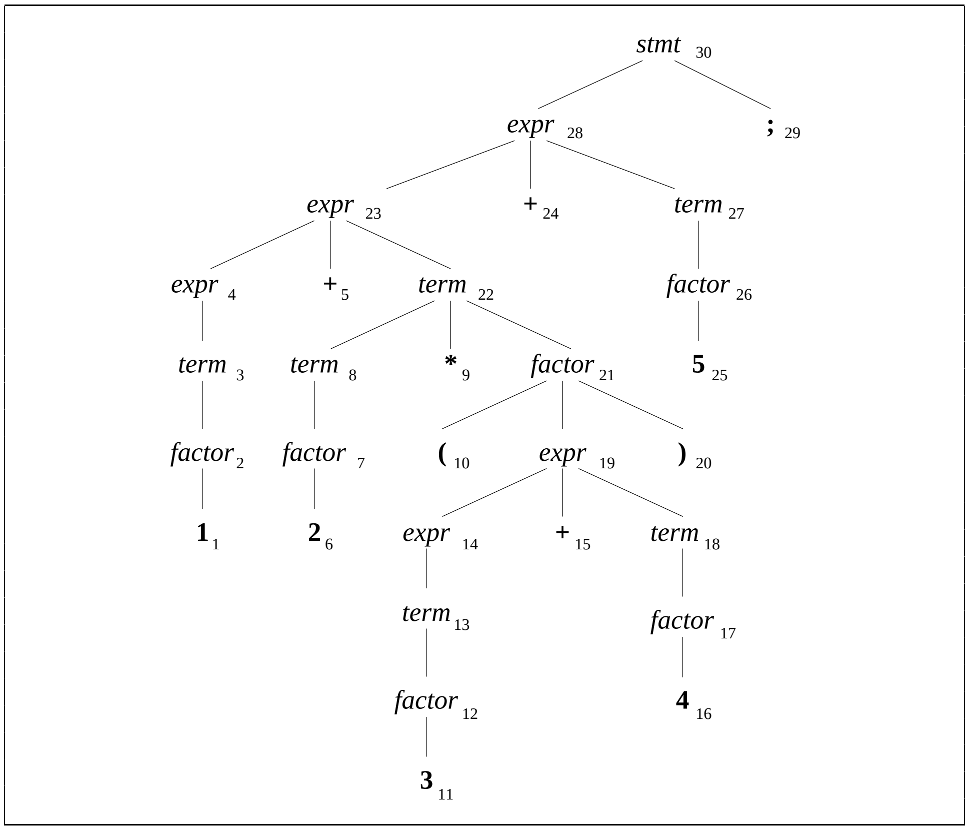
	No Separator	
	Right associative	**Left associative**
At least one element	*list* → MEMBER *list* | MEMBER	*list* → *list* MEMBER | MEMBER
Zero elements okay	*list* → MEMBER *list* | ε	*list* → *list* MEMBER | ε

	Separator Between List Elements	
	Right associative	**Left associative**
At least one element	*list* → MEMBER **delim** *list* | MEMBER	*list* → *list* **delim** MEMBER | MEMBER
Zero elements okay	*opt_list* → *list* | ε *list* → MEMBER **delim** *list* | MEMBER	*opt_list* → *list* | ε *list* → *list* **delim** MEMBER | MEMBER

A MEMBER is a list element; it can be a terminal, a nonterminal, or a collection of terminals and nonterminals. If you want the list to be a list of terminated objects such as semicolon-terminated declarations, MEMBER should take the form: MEMBER→ α **TERMINATOR**, where α is a collection of one or more terminal or nonterminal symbols.

Page 181 - *Figure 3.6. Change all "statements" to "stmt" for consistency. Also change "expression" to "expr". A new figure follows:*

Figure 3.6. A Parse Tree for `1+2*(3+4)+5;`

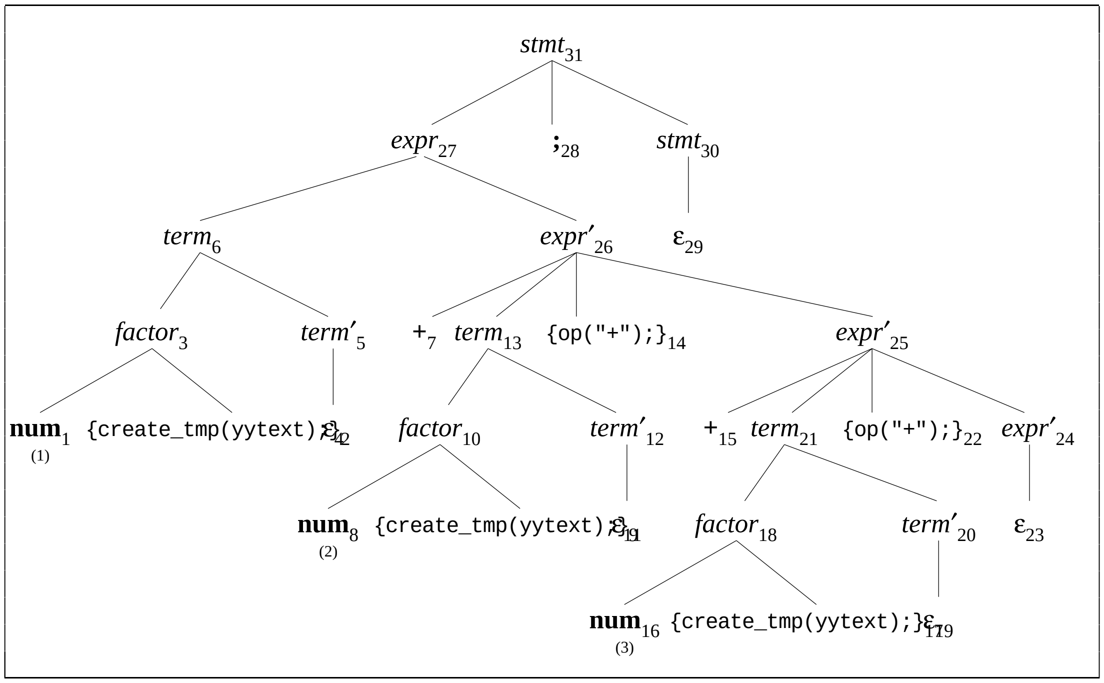


Page 183 - *Display at bottom of page. Remove the exclamation point. The expression should read:*

expr′ → + *term* `{op('+');}` *expr′*

Page 186 - *Figure 3.8. Change caption to "Augmented parse tree for* `1+2+3;`*" and change "statements" to "stmt". A new figure follows:*

Figure 3.8. Augmented Parse Tree for `1+2+3;`



Page 188 - *The caption for Listing 3.4 should say "Inherited" (not "Synthesized") "Attributes." A replacement caption follows:*

Listing 3.4. *naive.c*— Code Generation with Inherited Attributes

Page 190 - *First three lines beneath the figure (which start "right-hand side") should read:*

right-hand side of the production, as if they were used in the subroutine. For example, say that an attribute, `t`, represents a temporary variable name, and it is attached to an *expr*; it is represented like this in the subroutine representing the *expr:*

Page 196 - *Figure 4.1. both instances of the word "number" should be in boldface.*

number **number**

Page 208 - *Third and eighth lines change "automata" to "automaton". Replacement lines:*

solution is a push-down automaton in which a state machine controls the activity on the ‘
‘

The tables for the push-down automaton used by the parser are relatively straightfor-
‘ ‘

Page 210 - *Figure 4.6,* `Yy_d[factor][LP]` should be "9," not "8."
9

Page 213 - *First Paragraph, add* **SEMICOLON** *to the list on the third line. A replacement paragraph follows.*
Production 1 is applied if a *statement* is on top of the stack and the input symbol is an **OPEN_CURLY.** Similarly, Production 2 is applied when a *statement* is on top of the stack and the input symbol is an **OPEN_PAREN**, **NUMBER**, **SEMICOLON**, or **IDENTIFIER** (an **OPEN_PAREN** because an *expression* can start with an **OPEN_PAREN** by Production 3, a **NUMBER** or **IDENTIFIER** because an *expression* can start with a *term,* which can, in turn, start with a **NUMBER** or **IDENTIFIER**. The situation is complicated when an *expression* is on top of the stack, however. You can use the same rules as before to figure out whether to apply Productions 3 or 4, but what about the ε production (Production 5)? The situation is resolved by looking at the symbols that can follow an *expression* in the grammar. If *expression* goes to ε, it effectively disappears from the current derivation (from the parse tree)—it becomes transparent. So, if an *expression* is on top of the stack, apply Production 5 if the current lookahead symbol can follow an *expression* (if it is a **CLOSE_CURLY**, **CLOSE_PAREN**, or **SEMICOLON**). In this last situation, there would be serious problems if **CLOSE_CURLY** could also start an *expression*. The grammar would not be LL(1) were this the case.

Page 213 - *Last six lines should be replaced with the following seven lines.*

ones. Initially, add those terminals that are at the far left of a right-hand side:

FIRST(stmt)	=	{ }
FIRST(expr)	=	{ε}
FIRST(expr′)	=	{**PLUS**, ε}
FIRST(term)	=	{ }
FIRST(term′)	=	{**TIMES**, ε}
FIRST(factor)	=	{**LEFT_PAREN**, **NUMBER**}

Page 214 - *Remove the first line beneath Table 4.14 [which starts ‘FIRST(factor)’’].*

Page 214 - *Table 4.13, item (3), third line. Replace ‘is are’’ with ‘are.’’ A replacement table follows:*

Table 3.3. Finding FIRST Sets

(1)	FIRST(**A**), where **A** is a terminal symbol, is {**A**}. If A is ε, then ε is put into the FIRST set.
(2)	Given a production of the form $s \rightarrow \mathbf{A}\ \alpha$ where *s* is a nonterminal symbol, **A** is a terminal symbol, and α is a collection of zero or more terminals and nonterminals, **A** is a member of FIRST(s).
(3)	Given a production of the form $s \rightarrow b\ \alpha$ where *s* and *b* are single nonterminal symbols, and α is a collection of terminals and nonterminals, everything in FIRST(b) is also in FIRST(s) . This rule can be generalized. Given a production of the form: $s \rightarrow \alpha\ B\ \beta$ where *s* is a nonterminal symbol, α is a collection of zero or more nullable nonterminals,† *B* is a single terminal or nonterminal symbol, and β is a collection of terminals and nonterminals, then FIRST(s) includes the union of FIRST(*B*) and FIRST(α). For example, if α consists of the three nullable nonterminals *x*, *y*, and *z*, then FIRST(s) includes all the members of FIRST(x), FIRST(y), and FIRST(z), along with everything in FIRST(*B*).

† A nonterminal is nullable if it can go to ε by some derivation. ε is always a member of a nullable nonterminal's FIRST set.

Page 214 - *This is a change that comes under the "should be explained better" category and probably won't make it into the book until the second edition. It confuses the issue a bit to put ε into the FIRST set, as per rule (1) in Table 4.13. (In fact, you could argue that it shouldn't be there at all.) I've put ε into the FIRST sets because its presence makes it easier to see if a production is nullable (it is if ε is in the FIRST set). On the other hand, you don't have to transfer the ε to the FOLLOW set when you apply the rules in Table 4.15 because ε serves no useful purpose in the FOLLOW set. Consequently, ε doesn't appear in any of the FOLLOW sets that are derived on pages 215 and 216.*

Page 214 - *Bottom line and top of next page. Add ε to FIRST(expr), FIRST(expr′), and FIRST(term′). A replacement display, which replaces the bottom two lines of page 214 and the top four lines of page 215, follows:*

FIRST(stmt)	=	{**LEFT_PAREN**, **NUMBER**, **SEMI**}
FIRST(expr)	=	{**LEFT_PAREN**, **NUMBER**, ε}
FIRST(expr′)	=	{**PLUS**, ε}
FIRST(term)	=	{**LEFT_PAREN**, **NUMBER**}
FIRST(term′)	=	{**TIMES**, ε}
FIRST(factor)	=	{**LEFT_PAREN**, **NUMBER**}

Page 216 - *Add the following sentence to the end of item (2):*

Note that, since ε serves no useful purpose in a FOLLOW set, it does not have to be transfered from the FIRST to the FOLLOW set when the current rule is applied. A replacement table follows:

Table 3.4. Finding FOLLOW Sets

(1)	If *s* is the goal symbol, eoi (the end-of-input marker) is in FOLLOW(s);
(2)	Given a production of the form: $s \rightarrow \ldots a\, B \ldots$ where *a* is a nonterminal and *B* is either a terminal or nonterminal, FIRST(*B*) is in FOLLOW(a); To generalize further, given a production of the form: $s \rightarrow \ldots a\, \alpha\, B \ldots$ where *s* and *a* are nonterminals, α is a collection of zero or more nullable nonterminals and *B* is either a terminal or nonterminal. FOLLOW(a) includes the union of FIRST(α) and FIRST(*B*). Note that, since ε serves no useful purpose in a FOLLOW set, it does not have to be transfered from the FIRST to the FOLLOW set when the current rule is applied.
(3)	Given a production of the form: $s \rightarrow \ldots a$ where *a* is the rightmost nonterminal on the right-hand side of a production, everything in FOLLOW(s) is also in FOLLOW(a). (I'll describe how this works in a moment.) To generalize further, given a production of the form: $s \rightarrow \ldots a\, \alpha$ where *s* and *a* are nonterminals, and α is a collection of zero or more nullable nonterminals, everything in FOLLOW(s) is also in FOLLOW(a).

Page 217 - *Grammar in the middle of the page. Delete the* pk *and* adm *at the right edges of Productions 1 and 2.*

Page 218 - *Move the last line of page 217 to the top of the current page to eliminate the orphan.*

Page 218 - *Table 4.16, Replace the table with the following one (I've made several small changes). You may also want to move the widow at the top of the page to beneath the table while you're at it.*

Table 4.16. Finding LL(1) Selection Sets

(1) A production is *nullable* if the entire right-hand side can go to ε. This is the case, both when the right-hand side consists only of ε, and when all symbols on the right-hand side can go to ε by some derivation.

(2) **For nonnullable productions:** Given a production of the form

$$s \rightarrow \alpha\, B \ldots$$

where *s* is a nonterminal, α is a collection of one or more nullable nonterminals, and *B* is either a terminal or a nonnullable nonterminal (one that can't go to ε) followed by any number of additional symbols: the LL(1) select set for that production is the union of FIRST(α) and FIRST(*B*). That is, it's the union of the FIRST sets for every nonterminal in α plus FIRST(*B*). If α doesn't exist (there are no nullable nonterminals to the left of *B*), then SELECT(*s*)=FIRST(*B*).

(3) **For nullable productions:** Given a production of the form

$$s \rightarrow \alpha$$

where *s* is a nonterminal and α is a collection of zero or more nullable nonterminals (it can be ε): the LL(1) select set for that production is the union of FIRST(α) and FOLLOW(s). In plain words: if a production is nullable, it can be transparent—it can disappear entirely in some derivation (be replaced by an empty string). Consequently, if the production is transparent, you have to look through it to the symbols that can follow it to determine whether it can be applied in a given situation.

Page 223 - *Replace the last two lines on the page as follows:*
Ambiguous productions, such as those that have more than one occurrence of a given nonterminal on their right-hand side, cause problems in a grammar because a unique parse tree is not generated for a given input. As we've seen, left factoring can be used to

Page 224 - *The sentence on lines 13 and 14 (which starts with "If an ambiguous") should read as follows:*

If an ambiguous right-hand side is one of several, then all of these right-hand sides must move as part of the substitution. For example, given:

Page 228 - *8th line from the bottom, the "Y" in "You" should be in lower case.*

you can use a corner substitution to make the grammar self-recursive, replacing the ‘ ‘

Page 222 - *First line of text should read "Figures 4.5 and 4.6" rather than "Figure 4.6" A replacement paragraph follows:*
4.5 are identical in content to the ones pictured in Figures 4.5 and 4.6 on page 210. Note that the `Yyd` table on lines 179 to 184 is not compressed because this output file was generated with the *−f* switch active. Were *−f* not specified, the tables would be pair compressed, as is described in Chapter Two. The `yy_act()` subroutine on lines 199 to 234 contains the `switch` that holds the action code. Note that `$` references have been translated to explicit value-stack references (`Yy_vsp` is the value-stack pointer). The `Yy_synch` array on lines 243 to 248 is a −1-terminated array of the synchronization tokens specified in the `%synch` directive.

Page 237 - *The loop control on line 377 of Listing 4.6 won't work reliably in the 8086 medium or compact models. Replace lines 372–384 with the following:*

```
372       int   nterms;               /* # of terms in the production  */
373       start = Yy_pushtab[ production ];
374       for( end = start; *end; ++end )     /* After loop, end is positioned */
375       ;               /* to right of last valid symbol */
376       count = sizeof(buf);
377       *buf  = '\0';
378       for(nterms = end - start; --nterms >= 0 && count > 0 ; )    /* Assemble */
379       {                                /* string.  */
380       strncat( buf, yy_sym(*--end), count );
381       if( (count -= strlen( yy_sym(*end) + 1 )) < 1 )
382           break;
383       strncat( buf, " ", --count );
384       }
```

Page 242 - *Pieces of the section heading for Section 4.9.2 are in the wrong font, and the last two lines are messed up. Replace with the following:*

4.9.2 Occs and LLama Debugging Support—*yydebug.c*

This section discusses the debug-mode support routines used by the llama-generated parser in the previous section. The same routines are used by the occs-generated parser discussed in the next chapter. You should be familiar with the interface to the *curses*, window-management functions described in Appendix A before continuing.

Page 255 - *Fourth line beneath the listing (starting with "teractive mode"), replace comma following the close parenthesis with a period. The line should read:*

teractive mode (initiated with an *n* command). In this case, a speedometer readout that ‘
‘

Page 271-303 - *Odd numbered pages. Remove all tildes from the running heads.*

Page 274 - *Add the statement* `looking_for_brace = 0;` between lines 179 and 180. Do it by replacing lines 180–190 with the following:

```
180                         looking_for_brace = 0;
181                     }
182                     else
183                     {
184                         if( c == '%' ) looking_for_brace = 1;
185                         else       output( "%c", c );
186                     }
187                     }
188                 }
189                 return CODE_BLOCK;
190               }
```

Page 278 - *Fifth line from bottom. Replace* `{create_tmp(yytext);}` with the following (to get the example to agree with Figure 4.9):

```
{rvalue(yytext);}
```

Page 282 - *The last paragraph should read as follows (the Listing and Table numbers are wrong):*

The recursive-descent parser for LLama is in Listing 4.25. It is a straightforward representation of the grammar in Table 4.19.

Page 312 - *Listing 4.30. Replace lines 60 to 63 with the following:*

```
60  PRIVATE int *Dtran;     /* Internal representation of the parse table.
61                           * Initialization in make_yy_dtran() assumes
62                           * that it is an int [it calls memiset()].
63                           */
```

Page 315 - *Listing 4.30. Replace lines 231 to 240 with the following:*

```
231     nterms    = USED_TERMS + 1;         /* +1 for EOI */
232     nnonterms = USED_NONTERMS;
233
234     i = nterms * nnonterms;         /* Number of cells in array */
235
236     if( !(Dtran = (int *) malloc(i * sizeof(*Dtran)) ))  /* number of bytes */
237     ferr("Out of memory\n");
238
239     memiset( Dtran, -1, i );            /* Initialize Dtran to all failures */
240     ptab( Symtab, fill_row, NULL, 0 );  /* and fill nonfailure transitions. */
```

Page 330 - *Listing 4.34, line 464. Delete everything except the line number.*

Page 330 - *Last two lines of second paragraph should read as follows:*
bottom-up parse tables are created, below. Most practical LL(1) grammars are also LR(1) grammars, but not the other way around.

Page 330 - *Add the right-hand side "*| **NUMBER***" to the grammar in exercise 4.5. Also, align the table in Exercise 4.5 under the text so that it no longer extends into the gutter.*

$$
\begin{array}{lcl}
\textit{expr} & \rightarrow & -\ \textit{expr} \\
 & | & \textbf{*}\ \textit{expr} \\
 & | & \textit{expr}\ \textbf{*}\ \textit{expr} \\
 & | & \textit{expr}\ /\ \textit{expr} \\
 & | & \textit{expr} = \textit{expr} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textbf{(}\ \textit{expr}\ \textbf{)} \\
 & | & \textbf{NUMBER}
\end{array}
$$

Page 349 - *Replace Table 5.5 with the following table:*

Table 5.5. Error Recovery for `1++2`

Stack					Input	Comments
state:	–				`1 + + 2 |-`	*Shift start state*
parse:	–					
state:	0				`1 + + 2 |-`	*Shift NUM (goto 1)*
parse:	$					
state:	0	1			`+ + 2 |-`	*Reduce by Production 3 (!T→NUM)*
parse:	$	NUM				*(Return to 0, goto 3)*
state:	0	3			`+ + 2 |-`	*Reduce by Production 2 (!E→!T)*
parse:	$	!T				*(Return to 0, goto 2)*
state:	0	2			`+ + 2 |-`	*Shift !+ (goto 4)*
parse:	$	!E				
state:	0	2	4		`+ 2 |-`	*ERROR (no transition in table)*
parse:	$	!E	!+			*Pop one state from stack*
state:	0	2			`+ 2 |-`	*There is a transition from 2 on !+*
parse:	$	!E				*Error recovery is successful*
state:	0	2			`+ 2 |-`	*Shift !+ (goto 4)*
parse:	$	!E				
state:	0	2	4		`2 |-`	*Shift NUM (goto 1)*
parse:	$	!E	!+			
state:	0	2	4	1	`2 |-`	*Shift NUM (goto 1)*
parse:	$	!E	!+	NUM		
state:	0	2	4	1	`|-`	*Reduce by Production 3 (!T→NUM)*
parse:	$	!E	!+	NUM		*(Return to 4, goto 5)*
state:	0	2	4	5	`|-`	*Reduce by Production 1 (!E→!E!+!T)*
parse:	$	!E	!+	!T		*(Return to 0, goto 2)*
state:	0	2			`|-`	*Accept*
parse:	$	!E				

Page 360 - *Figure 5.6. The item immediately below the line in State 7 (ie. the first closure item) should be changed from !E→.!E!*!F to !T→.!T!*!F*

!T → . !T !* !F

Page 361 - *Third paragraph of section 5.6.2 (which starts "The FOLLOW"). Replace the paragraph with the following one:*

The FOLLOW sets for our current grammar are in Table 5.6. Looking at the shift/reduce conflict in State 4, FOLLOW(!E) doesn't contain a !*, so the SLR(1) method works in this case. Similarly, in State 3, FOLLOW(s) doesn't contain a !+, so everything's okay. And finally, in State 10, there is an outgoing edge labeled with a !*, but FOLLOW(!E) doesn't contain a !*. Since the FOLLOW sets alone are enough to resolve the shift/reduce conflicts in all three states, this is indeed an SLR(1) grammar.

Page 361 - *First paragraph in section 5.6.3 (which starts "Continuing our quest"). Replace the paragraph with the following one:*

Some symbols in FOLLOW set are not needed.

Lookahead set.

Many grammars are not as tractable as the current one—it's likely that a FOLLOW set will contain symbols that also label an outgoing edge. A closer look at the machine yields an interesting fact that can be used to solve this difficulty. A nonterminal's FOLLOW set includes *all* symbols that can follow that nonterminal in every possible context. The state machine, however, is more limited. You don't really care which symbols can follow a nonterminal in every possible case; you care only about those symbols that can be in the input when you reduce by a production that has that nonterminal on its left-hand side. This set of relevant lookahead symbols is typically a subset of the complete FOLLOW set, and is called the *lookahead set*.

Page 362 - *Ninth line from bottom. Delete 'only." A replacement for this, and the following four lines follows:*

The process of creating an LR(1) state machine differs from that used to make an LR(0) machine only in that LR(1) items are created in the closure operation rather than LR(0) items. The initial item consists of the start production with the dot at the far left and |- as the lookahead character. In the grammar we've been using, it is:

Page 362 - *Last line. Delete the period. The line should read: $x \rightarrow \gamma$*

Page 363 - *The* **C** *is in the wrong font in both the first marginal note and the first display (on the third line). It should be in Roman.*

[!S→α !. *x* β, **C**].

[*x*→!.γ, FIRST(β **C**)].

Page 363 - *Ninth line from bottom. Replace 'new machine" with 'new states." A replacement paragraph follows:*

The process continues in this manner until no more new LR(1) items can be created. The next states are created as before, adding edges for all symbols to the right of the dot and moving the dots in the kernel items of the new states. The entire LR(1) state machine for our grammar is shown in Figure 5.7. I've saved space in the Figure by merging together all items in a state that differ only in lookaheads. The lookaheads for all such items are shown on a single line in the right column of each state. Figure 5.8

shows how the other closure items in State 0 are derived. Derivations for items in States 2 and 14 of the machine are also shown.

Page 364 - *Figure 5.7. About 3 inches from the left of the figure and 1¾ inches from the bottom, a line going from the box marked 2 to a circle with a B in it is currently labeled "t e f* **NUM (**.*" Delete the* e.

Page 365 - *The fourth line below Figure 5.7 should read:*

best of both worlds. Examining the LR(1) machine in Figure 5.7, you are immediately ‘
‘

Page 365 - *Figure 5.7 (continued). The upper-case F in the second item of State 16 should be lower case.*

!T→ . !T !* !F

Page 366 - *First and third line under the Figure. The figure numbers are called out incorrectly in the text. The first three lines beneath the figure should read:*

parenthesis first. The outer part of the machine (all of the left half of Figure 5.7 except States 6 and 9) handles unparenthesized expressions, and the inner part (States 6 and 9, and all of the right half of Figure 5.7) handles parenthesized subexpressions. The parser
‘ ‘

Page 370 - *Listing 5.2, line 14. Change the sentence "*`Reduce by production n`*" to read as follows (leave the left part of the line intact):*

```
Reduce by production n,  n == -action.
```

Page 371 - *Listing 5.2, line 16. Change the sentence "*`Shift to state n`*" to read as follows (leave the left part of the line intact):*

```
Shift to state n,  n == action.
```

Page 373 - *Listing 5.4, line 6. Remove the* `yy` *in* `yylookahead`*. The corrected line looks like this:*

```
6      do_this = yy_next( Yy_action, state_at_top_of_stack(), lookahead );
```

Page 373 - *Listing 5.4, line 29. Change* `rhs_len` *to* `rhs_length`*. The corrected line looks like this:*

```
29          while( --rhs_length >= 0 )           /* pop rhs_length items */
```

Page 373 - *Last line. Change to read as follows (the state number is wrong):*

shifts to State 1, where the only legal action is a reduce by Production 6 if the next input
‘ ‘

Page 374 - *Paragraph beneath table, replace “you you” on third line with “you”. Entire replacement paragraph follows:*

There’s one final caveat. You cannot eliminate a single-reduction state if there is a code-generation action attached to the associated production because the stack will have one fewer items on it than it should when the action is performed—you won’t be able to access the attributes correctly. In practice, this limitation is enough of a problem that occs doesn’t use the technique. In any event, the disambiguating rules discussed in the next section eliminate many of the single-reduction states because the productions that cause them are no longer necessary.

Page 387 - *Listing 5.11, line 107. Add the word* `short`. The repaired line looks like this:

```
107   #define YYF ((YY_TTYPE)( (unsigned short)˜0 >>1 ))
```

Page 388 - *Second paragraph, third and fourth lines. Change “the largest positive integer” to “to the largest positive* `short int`.” and remove the following “I’m.” The repaired lines read as follows:

subroutine, `yy_act_next()`, which I’ll discuss in a moment.) It evaluates to the largest positive `short int` (with two’s complement numbers). Breaking the macro down: ‘
‘

Page 390 - *Listing 5.12, line 199. Change the sentence “`Reduce by production n`” to read as follows (leave the left part of the line intact):*

`Reduce by production n,  n == -action.`

Page 390 - *Listing 5.2, line 201. Change the sentence “`Shift to state n`” to read as follows (leave the left part of the line intact):*

`Shift to state n,  n == action.`

Page 397 - *Replace line 209 of Listing 5.14 with the following:*

```
209          YYD( yycomment("Popping %s from state stack\n", tos); )
```

Page 398 - *Listing 5.14, lines 219–222. Replace with the following code:*

```
219          Yy_vsp = Yy_vstack + (YYMAXDEPTH - yystk_ele(Yy_stack)) ;
220   #      ifdef YYDEBUG
221              yystk_p(Yy_dstack) = Yy_dstack +
222                      (YYMAXDEPTH - yystk_ele(Yy_stack));
```

Page 403 - *The loop control on Line 128 of Listing 5.15 doesn't work reliably in the 8086 compact or large model. To fix it, replace Line 97 of Listing 5.15 (p. 403) with the following (and also see change for next page):*

```
 97      int     i    ;
```

Page 404 - *Replace line 128 of Listing 5.15 with the following:*

```
128      for(i = (pp - prod->rhs) + 1; --i >= 0;  --pp )
```

Page 425 - *Lines 585–594. Replace with the following code:*

```
585      if( nclose )
586      {
587          assort( closure_items, nclose, sizeof(ITEM*), item_cmp );
588          nitems  = move_eps( cur_state, closure_items, nclose );
589          p       = closure_items + nitems;
590          nclose -= nitems ;
591
592          if( Verbose > 1 )
593          pclosure( cur_state, p, nclose );
594      }
```

Page 440 - *Listing 5.33, replace the code on lines 1435 to 1438 with the following (be sure that the quote marks on the left remain aligned with previous lines):*

```
1435      "   action <  0   -- Reduce by production n,  n == -action.",
1436      "   action == 0   -- Accept. (ie. Reduce by production 0.)",
1437      "   action >  0   -- Shift to state n,  n == action.",
1438      "   action == YYF -- error.",
```

Page 447 - *Line 14. Change "hardly every maps" to "hardly ever maps". The line should read:*

ideal machine hardly ever maps to a real machine in an efficient way, so the generated ‘ ‘

Page 452 - *First line below Listing 6.2. Change “2048” to “1024”. Line should read:*

In addition to the register set, there is a 1024-element, 32-bit wide stack, and two ‘ ‘

Page 453 - *Figure 6.1. Around the middle of the figure. Change “2048” to “1024”. Line should read:*

1,024
32-bit
lwords

Page 466 - *Listing 6.11, lines 53 and 54, change* `sp` to `__sp` (two underscores).

```
53  #define push(n)         (--__sp)->l = (lword)(n)
54  #define pop(t)      (t)( (__sp++)->l )
```

Page 471 - *Ninth line. Replace with* `fp+4` with `fp+16`. Replace the last four lines of the paragraph with the following lines:
`call()` subroutine modifies `wild`, it just modifies the memory location at `fp+16`, and on the incorrect stack, ends up modifying the return address of the calling function. This means that `call()` could work correctly, as could the calling function, but the program would blow up when the calling function returned.

Page 475 - *Listing 6.19, Line 80. Replace the first* `(b)` with a `(s)`. The line should now read:

```
80  #define BIT(b,s)    if( (s) & (1 << (b)) )
```

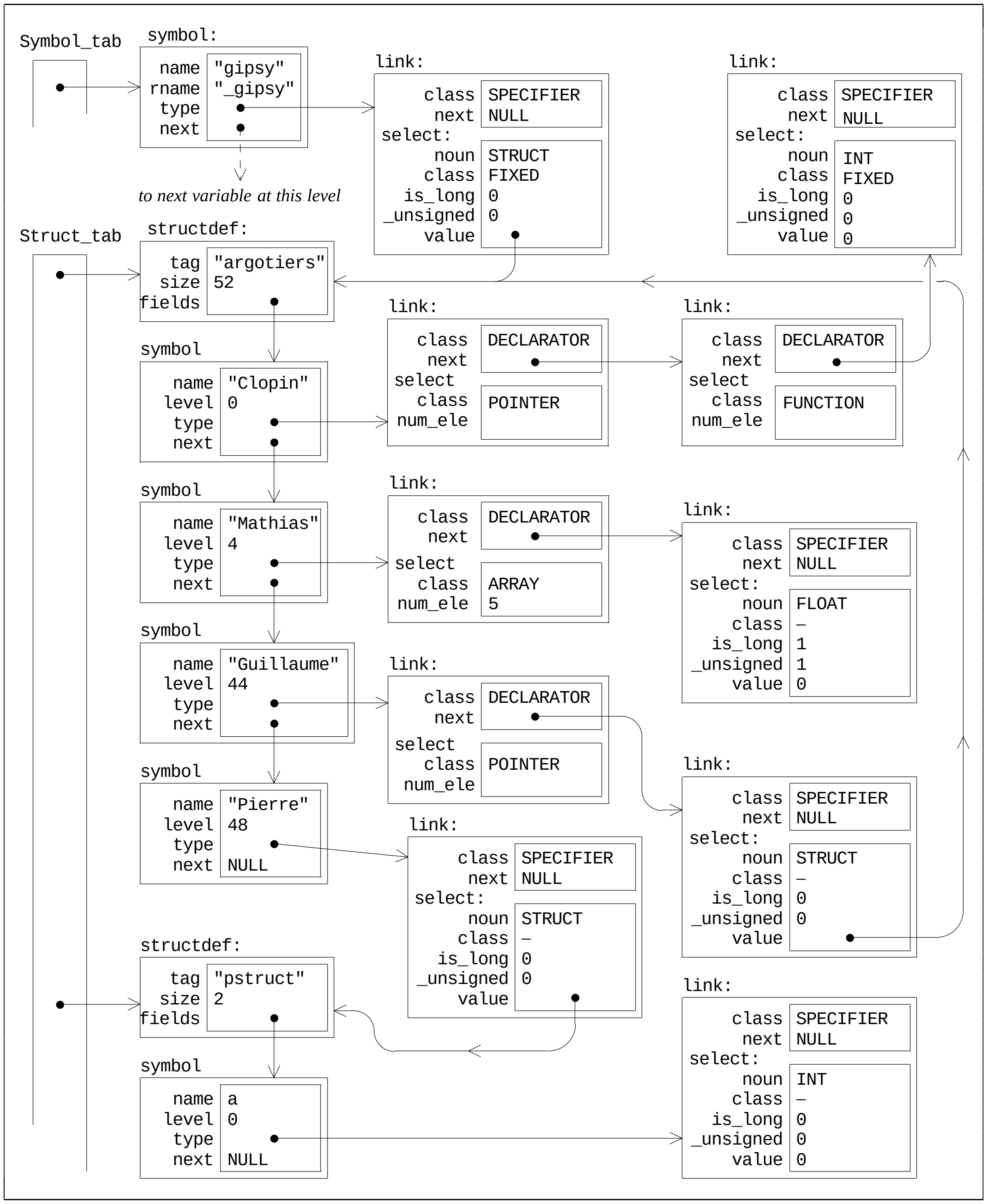
Page 486 - *Replace last line of first paragraph (which now reads ‘6.10”) with the following:*
6.11.

Page 494 - *Listing 6.25, replace lines 129–142 with the following:*

```
129  #define IS_SPECIFIER(p)  ((p) && (p)->class==SPECIFIER )
130  #define IS_DECLARATOR(p) ((p) && (p)->class==DECLARATOR )
131  #define IS_ARRAY(p)   ((p) && (p)->class==DECLARATOR && (p)->DCL_TYPE==ARRAY   )
132  #define IS_POINTER(p) ((p) && (p)->class==DECLARATOR && (p)->DCL_TYPE==POINTER )
133  #define IS_FUNCT(p)   ((p) && (p)->class==DECLARATOR && (p)->DCL_TYPE==FUNCTION)
134  #define IS_STRUCT(p)  ((p) && (p)->class==SPECIFIER  && (p)->NOUN == STRUCTURE )
135  #define IS_LABEL(p)   ((p) && (p)->class==SPECIFIER  && (p)->NOUN == LABEL     )
136
137  #define IS_CHAR(p)      ((p) && (p)->class == SPECIFIER && (p)->NOUN == CHAR )
138  #define IS_INT(p)       ((p) && (p)->class == SPECIFIER && (p)->NOUN == INT  )
139  #define IS_UINT(p)  (   IS_INT(p) && (p)->UNSIGNED       )
140  #define IS_LONG(p)      (   IS_INT(p) && (p)->LONG           )
141  #define IS_ULONG(p) (   IS_INT(p) && (p)->LONG && (p)->UNSIGNED  )
142  #define IS_UNSIGNED(p)  ((p) && (p)->UNSIGNED                )
```

Page 496 - *Figure 6.13. All of the* `link` structures that are labeled `SPECIFIER` should be labeled `DECLARATOR` and vice versa. The corrected figure follows:

Figure 6.13. Representing a Structure in the Symbol Table



Page 500 - *First sentence below figure should start “The subroutine”*

The subroutine in Listing 6.29 manipulates declarators: `add_declarator()` adds
‘ ‘

Page 503 - *Listing 6.31, line 281, should read as follows:*

```
281            (p1->DCL_TYPE==ARRAY && (p1->NUM_ELE != p2->NUM_ELE)) )
```

Page 520 - *Replace line 71 of Listing 6.37 with the following line:*

```
71  (’.’)|(’\\.’)|(’\\{o}({o}{o}?)?’)|(’\\x{h}({h}{h}?)?’)    |
```

Replace line 76 of Listing 6.37 with the following line:

```
76  ({d}+|{d}+\.{d}*|{d}*\.{d}+)([eE][\-+]?{d}+)?[fF]?   return FCON ;
```

Page 521 - *Listing 6.37, Lines 138–143. Replace as follows:*

```
138  typedef struct      /* Routines to recognize keywords. A table  */
139  {           /* lookup is used for this purpose in order to  */
140      char  *name;    /* minimize the number of states in the FSM. A  */
141      int   val;      /* KWORD is a single table entry.       */
142  }
143  KWORD;
```

Page 524 - *Second line of last paragraph, remove period after* **TYPE** *and change “Listing 6.38” to Listing 6.39. The repaired line should read as follows:*

reduces *type_specifier*→**TYPE** (on line 229 of Listing 6.39). The associated action ‘
‘

Page 527 - *First line below Listing 6.40. Change “Listing 6.40” to Listing 6.39. The repaired line should read as follows:*

216 to 217 of Listing 6.39.) There are currently three attributes of interest: `$1` and `$$` ‘
‘

Page 553 - *Listing 6.56, line 10. Change* `L0` to `(L0*4)`.

```
10   #define  T(n) (fp-(L0*4)-(n*4))
```

Page 556 - *Listing 6.58, line 539. Change* `%s` to (%s*4)

```
539          yycode(   "#define  T(n) (fp-(%s*4)-(n*4))\n\n", Vspace );
```

Page 558 - *Listing 6.60, lines 578–582. Replace with the following:*

```
578          discard_link_chain(existing->type); /* Replace existing type      */
579          existing->type   = sym->type;   /* chain with the current one.*/
580          existing->etype  = sym->etype;
581          sym->type = sym->etype = NULL;  /* Must be NULL for discard_- */
582      }                   /* symbol() call, below.      */
```

Page 558 - *Listing 6.60, lines 606 and 607.* `i` is not used and the initial offset should be 8. Replace lines 606 and 607 with the following:

```
606      int  offset = 8;        /* First parameter is always at BP(fp+8):      */
607                 /* 4 for the old fp, 4 for the return address. */
```

Page 560 - *Page 560, Listing 6.61. Replace lines 578–580 with the following:*

```
578      : LC              {   if( ++Nest_lev == 1 )
579                            loc_reset();
580                        }
```

Page 573 - *Fifth line from the bottom. Insert a period after "needed". The line should read:*
needed. The stack is shrunk with matching additions when the variable is no longer ‘ ‘

Page 574 - *Figure 6.18, the* `L0` in the **`#define`** `T(n)` should be `(L0*4)`.

```
#define T(n)  (fp-(L0*4)-(n*4))
```

Page 578 - *First paragraph. There’s an incomplete sentence on the first line. Replace with the following paragraph and add the marginal note:*

Marking a stack cell as “in use”.

The cell is marked as “in use” on line 73. The `Region` element corresponding to the first cell of the allocated space is set to the number of stack elements that are being allocated. If more than one stack element is required for the temporary, adjacent cells that are part of the temporary are filled with a place marker. Other subroutines in Listing 6.64 de-allocate a temporary variable by resetting the equivalent `Region` elements to zero, de-allocate all temporary variables currently in use, and provide access to the high-water mark. You should take a moment and review them now.

Page 590 - *Listing 6.69, line 214. Replace with the following:*

```
214           case CHAR:       return BYTE_PREFIX;
```

Page 598 - *Third paragraph, second line (which starts “type* `int` for”), replace line with the following one:

type `int` for the undeclared identifier (on line 62 of Listing 6.72). The ‘
‘

Page 601 - *First line beneath Listing 6.76. Replace “generate” with “generated”:*

So far, none of the operators have generated code. With Listing 6.77, we move ‘
‘

Page 607 - *Listing 6.84, line 328: Change the* `||` to an `&&`.

```
328       if( !IS_INT(offset->type) && !IS_CHAR(offset->type) )
```

Page 608 - *Fonts are wrong in all three marginal notes. Replace them with the ones given here.*

Operand to `*` or `[]` must be array or pointer.

Attribute synthesized by `*` and `[]` operators.

Rules for forming lvalues and rvalues when processing `*` and `[]`.

Page 613 - *First line of last paragraph is garbled. Since the fix affects the entire paragraph, an entire replacement paragraph follows. Everything that doesn’t fit on page 613 should be put at the top of the next page.*

`call()`

unary→**NAME**

The `call()` subroutine at the top of Listing 6.87 generates both the `call` instruction and the code that handles return values and stack clean up. It also takes care of implicit subroutine declarations on lines 513 to 526. The action in *unary*→**NAME** creates a `symbol` of type `int` for an undeclared identifier, and this symbol eventually ends up here as the incoming attribute. The `call()` subroutine changes the type to "function returning int" by adding another `link` to the head of the type chain. It also clears the `implicit` bit to indicate that the symbol is a legal implicit declaration rather than an undeclared variable. Finally, a C-code `extern` statement is generated for the function.

Page 617 - *Listing 6.87, line 543. Replace* `nargs` with `nargs * SWIDTH`.

```
543         gen( "+=%s%d" , "sp", nargs * SWIDTH );  /* sp is a byte pointer, */
```

Page 619 - *Listing 6.88, line 690. Delete the* `->name`. The repaired line should look like this:

```
690                         gen( "EQ",       rvalue( $1 ), "0" );
```

Disk only. *Page 619, Listing 6.88. Added semantic-error checking to first (test) clause in ?: operator. Tests to see if its an integral type. Insert the following between lines 689 and 690:*

```
                            if( !IS_INT($1->type) )
                               yyerror("Test in ?: must be integral\n");
```

Page 619 - *Listing 6.88. Replace line 709 with the following line:*

```
709                         gen( "=",     $$->name,   rvalue($7) );
```

Page 644 - *Lines 895 and 896. There is a missing double-quote mark on line 895, insertion of which also affects the formatting on line 896. Replace lines 895 and 896 with the following:*

```
895                    gen("goto%s%d", L_BODY,      $5 );
896                    gen(":%s%d",    L_INCREMENT, $5 );
```

Page 648 - *Listing 6.107, line 1, change the* `128` to `256`.

```
 1  #define CASE_MAX 256        /* Maximum number of cases in a switch */
```

Page 649 - *Listing 6.108. Add the following two lines between lines 950 and 951: (These lines will not have numbers on them, align the first* `p` in `pop` with the `g` in `gen_stab`... on the previous line.)

```
            pop( S_brk );
            pop( S_brk_label );
```

Page 658 - *First paragraph of section 7.2.1 should be replaced with the following one:*

A *strength reduction* replaces an operation with a more efficient operation or series of operations that yield the same result in fewer machine clock cycles. For example, multiplication by a power of two can be replaced by a left shift, which executes faster on most machines. (`x*8` can be done with `x<<3`.) You can divide a positive number by a power of two with a right shift (`x/8` is `x>>3` if x is positive) and do a modulus division by a power of two with a bitwise AND (`x%8` is `x&7`).

Page 671 - *Figure 7.1, third subfigure from the bottom. In initial printings, the asterisk that should be at the apex of the tree had dropped down about ½ inch. Move it up in these printings. In later printings, there are two asterisks. Delete the bottom one.*

Page 681 - *Listing A.1, lines 19 and 20 are missing semicolons. Change them to the following:*

```
19      typedef long     time_t; /* for the VAX, may have to change this */
20      typedef unsigned size_t; /* for the VAX, may have to change this */
```

Page 682 - *Listing A.1, line 52. Delete all the text on the line, but leave the asterisk at the far left.*

Page 688 - *6th line from the bottom. Remove "though" at start of line.*

it might introduce an unnecessary conversion if the stack type is an `int`, `short`, or `char`. These multiple type conversions will also cause portability problems if the `stack_err()` macro evaluates to something that won't fit into a `long` (like a `double`).

Page 690 - *Last line, "calling conventions" should not be hyphenated.*

Page 696 - *Listing A.4, line 40. The comment is wrong. The line should read as follows:*

```
 40   #define _DIFFERENCE 2   /* (x in s1) and (x not in s2)  */
```

Page 702 - *Listing A.6. Change lines 98–102 to the following:*

```
 98      /* Enlarge the set to "need" words, filling in the extra words with zeros.
 99       * Print an error message and exit if there's not enough memory.
100       * Since this routine calls malloc, it's rather slow and should be
101       * avoided if possible.
102       */
```

Page 706 - *Listing A.8, line 330. Change* `unsigned` to `int`:

```
330      int     ssize;  /* Number of words in src set   */
```

Page 713 - *Third line from the bottom. "nextsym( )" should be in the Courier font. Replace the last three lines on the page with the following:*

passing the pointer returned from `find_sym()` to `nextsym()`, which returns either a pointer to the next object or `NULL` if there are no such objects. Use it like this:

Page 719 - *Second line above Listing A.17, change "tree" to "table":*

cial cases. The `delsym()` function, which removes an arbitrary node from the table, is shown in Listing A.17.

Page 722 - *Listing A.19, line 221. Replace with:*

```
221      return (*User_cmp)( (void*)(*p1 + 1), (void*)(*p2 + 1) );
```

Page 729 - *Listing A.26, line 50. Delete the* `(two required)` at the end of the line.

Page 736 - *Listing A.33, line 34. Change to the following:*

```
34   PUBLIC void stop_prnt(){}
```

Page 737 - *Listing A.33, line 97. Change to the following:*

```
97   char *str, *fmt, *argp;
```

Page 739 - *The swap statement in the code in the middle of the page is incorrect.*
Here is a replacement display:

```
int array[ ASIZE ];
int i, j, temp ;

for( i = 1; i < ASIZE; ++i )
    for( j = i-1; j >= 0; --j )
    if( array[j] > array[j+1] )
        swap( array[j], array[j+1] );
```

Page 743 - *Listing A.36. Delete the text (but not the line number) on line 4 (which now says* `#include <fcntl.h>`).

Page 745 - *Change caption of Listing A.38 to the following:*

Listing A.38. *memiset.c*— Initialize Array of `int` to Arbitrary Value

Page 755 - *Third line from the bottom. Delete the exclamation point. The line should read:*
images (25×80×2=4,000 bytes for the whole screen), and that much memory may not be
‘ ‘

Page 758 - *Listing A.45, line 49. Remove the semicolon at the far right of the line.*

Page 768 - *Listing A.61. Remove the exclamation point from the caption.*

Page 776 - *Line above heading for section A.11.2.2. Delete the* `void`.

Page 797 - *Replace Lines 50–59 of Listing A.84 with the following:*

```
        case '\b':  if( buf > sbuf )
                        {
                --buf;   wprintw( win, " \b" );
                }
                else
                {
                wprintw( win, " " );
                putchar('\007');
                }
                break;
```

Page 803 - *Just above heading for section B.2. The line should read:*

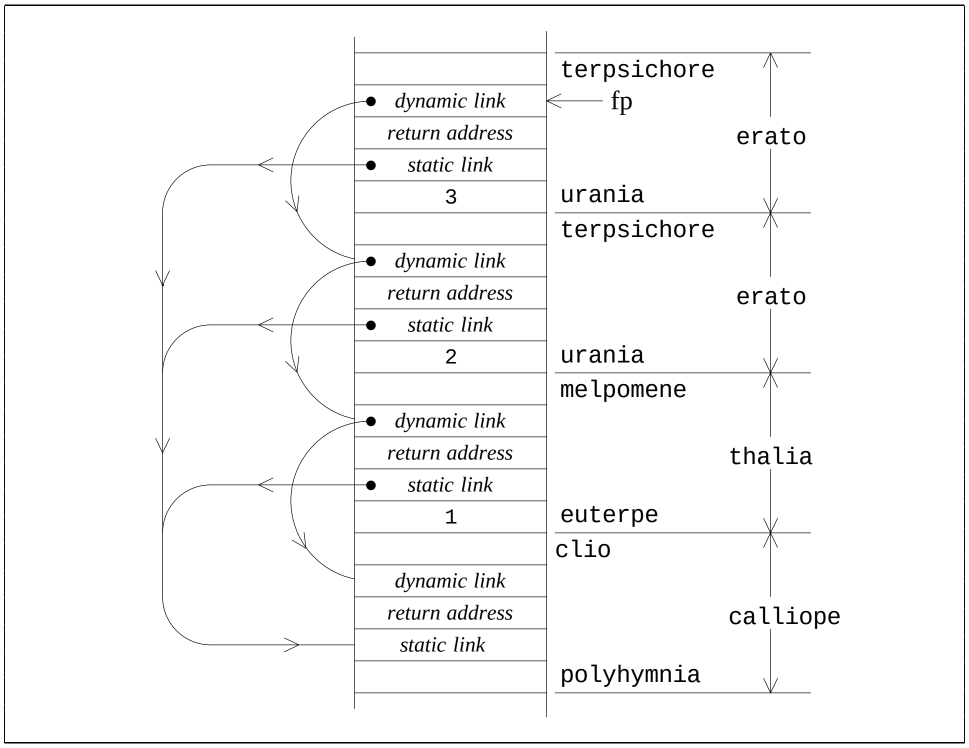
set to 100; otherwise, arg is set to 1.

Page 803 - *Replace the last paragraph on the page with the following one:*

The stack, when the recursive call to `erato` is active, is shown in Figure B.2. There is one major difference between these stack frames and C stack frames: the introduction of a second pointer called the *static link*. The *dynamic link* is the old frame pointer, just as in C. The static link points, not at the previously active subroutine, but at the parent subroutine in the nesting sequence—in the declaration. Since `erato` and `thalia` are both nested inside `calliope`, their static links point at `calliope`'s stack frame. You can chase down the static links to access the local variables in the outer routine's ' '

Page 804 - *Replace Figure B.2 with the following figure.*

Figure B.2. Pascal Stack Frames



Page 805 - *Top of the page. Replace the top seven lines (everything up to the paragraph that begins "This organization") with the following (you'll have to move the rest of the text on page 805 down to make room):*

stack frame. For example, `clio` can be accessed from `erato` with the following C-code:

```
r0.pp = WP(fp + 4);     /* r0 = static link */
_x    = W(r0.pp - 8);   /* x  = clio        */
```

You can access `polyhymnia` from `erato` with:

```
r0.pp = WP(fp + 4);     /* r0 = static link */
_x    = W(r0.pp + 8);   /* x  = clio        */
```

Though it's not shown this way in the current example, it's convenient for the frame pointer to point at the static, rather than the dynamic link to make the foregoing indirection a little easier to do. The static links can be set up as follows: Assign to each subroutine a *declaration level*, equivalent to the nesting level at which the subroutine is declared. Here, `calliope` is a level 0 subroutine, `erato` is a level 1 subroutine, and so forth. Then:

- **If a subroutine calls a subroutine at the same level**, the static link of the called subroutine is identical to the static link of the calling subroutine.
- **If a subroutine at level *N* calls a subroutine at level *N*+1**, the static link of the called subroutine points at the static link of the calling subroutine.
- **If a subroutine calls a subroutine at a lower (more outer) level**, use the following algorithm:

```
i = the difference in levels between the two subroutines;
p = the static link in the calling subroutine's stack frame;
while( --i >= 0 )
    p = *p;
the static link of the called subroutine = p;
```

Note that the difference in levels (`i`) can be figured at compile time, but you must chase down the static links at run time. Since the static link must be initialized by the calling subroutine (the called subroutine doesn't know who called it), it is placed beneath the return address in the stack frame.

Page 806 - *Change caption and title of Listing C.1 as follows:*

Listing C.1. A Summary of the C Grammar in Chapter Six.

Page 819 - *First full paragraph. Replace the "the the" on the fourth line with a single "the."*

A replacement paragraph follows:

The ˆ and $ metacharacters.

The ˆ and `$` metacharacters work properly in all MS-DOS input modes, regardless of whether lines end with `\r\n` or a single `\n`. Note that the newline is not part of the lexeme, even though it must be present for the associated expression to be recognized. Use `and\r\n` to put the end of line characters into the lexeme. (The `\r` is not required in UNIX applications, in fact it's an error under UNIX.) Note that, unlike the vi editor `ˆ$` does not match a blank line. You'll have to use an explicit search such as `\r\n\r\n` to find empty lines.

Page 821 - *Listing D.1, replace lines 14 and 15 with the following:*

```
14      while( yylex() )
15      ;
```

Page 821 - *First two lines beneath the listing. Delete both lines and replace with the following text:*

LeX and `yyleng` is adjusted accordingly. Zero is returned at end of file, –1 if the lexeme is too long.[13]

Page 821 - *Replace Footnote 13 at the bottom of the page with the one at the bottom of the page you are now reading.*

Page 828 - *Listing D.5.; in order to support the* `ul` suffix, replace line 16 with the following:

```
16  suffix  ([UuLl]|[uU][lL]) /* Suffix in integral numeric constant    */
```

Page 841 - *Replace the code on the first five lines of Listing E.2 with the following five lines:*

```
 1  %term ID    /*  an identifier   */
 2  %term NUM   /*  a number        */
 3  %left PLUS  /*  +           */
 4  %left STAR  /*  *           */
 5  %left LP RP /*  ( and )     */
```

Page 843 - *Paragraph starting -c[N],* , 2nd line. Delete “e” in "switche."

Page 860 - *15 lines from the bottom. Remove the underscores. The line should read:*

is from stack picture six to seven. `t0` and `t1` were put onto the stack when the rvalues ‘ ‘

[13] UNIX **lex** doesn’t return –1 and it doesn’t modify the `yytext` or `yyleng`; it just returns the next input character.

Page 861 - *Figure E.5. Remove all underscores. The figure should be replaced with the following one:*

Figure E.5. A Parse of A∗2

Page 862 - *Move caption for Listing E.9 to the left. (It should be flush with the left edge of the box.)*

Page 871 - *Figure E.6. Label the line between States 5 and 7 with* **STAR***.*

STAR

Page 880 - *Listing E.17, Lines 84 and 85, replace with the following:*

```
84      yycode("public word t0, t1, t2, t3;\n");
85      yycode("public word t4, t5, t6, t7;\n");
```

Page 886 - *Listing E.19. Replace lines 51 to 57 with the following:*

```
51  {0}   512, line 42 : {$1=$2=newname();}
52  {1}   513, line 42 : {freename($0);}
53  {2}   514, line 48 : {$1=$2=newname();}
54  {3}   515, line 49 : { yycode("%s+=%s\\n",$$,$0); freename($0); }
55  {4}   516, line 56 : {$1=$2=newname();}
56  {5}   517, line 57 : { yycode("%s*=%s\\n",$$,$0); freename($0);}
57  {6}   518, line 61 : { yycode("%s=%0.*s\\n",$$,yyleng,yytext); }
```

Page 887 - *Listing E.19. Replace lines 46 and 54 with the following:*

```
46              { yycode("%s+=%s\n",$$,$0); freename($0); } expr’
54              { yycode("%s*=%s\n",$$,$0); freename($0); } term’
```

Page 888 - *Listing E.19. Replace line 58 with the following:*

```
 58  factor  :   NUM_OR_ID { yycode("%s=%0.*s\n", $$, yyleng, yytext); }
```

Disk only. *I made several changes to searchen.c (p. 747) to make the returned path names more consistent (everything's now mapped to a* UNIX*-style name). Also added a disk identifier when running under DOS.*

Disk only. *Insert the following into the brace-processing code, between lines 115 and 116 of parser.lex* on page 273:

```
                if( i == '\n' && in_string )
                {
                lerror( WARNING,
                       "Newline in string, inserting \"\n");
                in_string = 0;
                }
```

Disk only. *The* `do_unop` subroutine on page 604 (Line 177 of Listing 6.79) wasn't handling incoming constant `values` correctly and it wasn't doing any semantic-error checking at all. It's been replaced by the following code. (Instructions are now generated only if the incoming `value` isn't a constant, otherwise the constant value at the end of the `link` chain is just modified by performing the indicated operation at compile time.)

```
177  value *do_unop( op, val )
178  int   op;
179  value *val;
180  {
181      char   *op_buf = "=?" ;
182      int    i;
183
184      if( op != '!' ) /* ~ or unary - */
185      {
186      if( !IS_CHAR(val->type) && !IS_INT(val->type) )
187          yyerror( "Unary operator requires integral argument\n" );
188
189      else if( IS_UNSIGNED(val->type) && op == '-' )
190          yyerror( "Minus has no meaning on an unsigned operand\n" );
191
192      else if( IS_CONSTANT( val->type ) )
193          do_unary_const( op, val );
194      else
195      {
196          op_buf[1] = op;
197          gen( op_buf, val->name, val->name );
198      }
199      }
200      else        /* ! */
201      {
202      if( IS_AGGREGATE( val->type ) )
203          yyerror("May not apply ! operator to aggregate type\n");
204
```

➡

Listing 5.10. continued…

```
    else if( IS_INT_CONSTANT( val->type ) )
        do_unary_const( '!', val );
    else
    {
        gen( "EQ",       rvalue(val), "0"            ); /* EQ(x, 0)       */
        gen( "goto%s%d", L_TRUE,      i = tf_label() ); /*   goto T000;   */
        val = gen_false_true( i, val );         /* fall thru to F */
    }
    }
    return val;
}
/*----------------------------------------------------------------------*/
do_unary_const( op, val )
int op;
value   *val;
{
    /* Handle unary constants by modifying the constant's value.  */

    link *t = val->type;

    if( IS_INT(t) )
    {
    switch( op )
    {
    case '~':   t->V_INT = ~t->V_INT;   break;
    case '-':   t->V_INT = -t->V_INT;   break;
    case '!':   t->V_INT = !t->V_INT;   break;
    }
    }
    else if( IS_LONG(t) )
    {
    switch( op )
    {
    case '~':   t->V_LONG = ~t->V_LONG; break;
    case '-':   t->V_LONG = -t->V_LONG; break;
    case '!':   t->V_LONG = !t->V_LONG; break;
    }
    }
    else
    yyerror("INTERNAL do_unary_const: unexpected type\n");
}
```

Disk only. *Page 506, Listing 6.32, lines 453–457: Modified subroutine* `type_str()` in file symtab.c to print the value of an integer constant. Replaced the following code:

```
if( link_p->NOUN != STRUCTURE )
    continue;
else
    i = sprintf(buf, " %s", link_p->V_STRUCT->tag ?
                link_p->V_STRUCT->tag : "untagged");
```

with this:

```
if( link_p->NOUN == STRUCTURE )
    i = sprintf(buf, " %s", link_p->V_STRUCT->tag ?
                link_p->V_STRUCT->tag : "untagged");

else if(IS_INT(link_p)   ) sprintf(buf, "=%d", link_p->V_INT  );
else if(IS_UINT(link_p)  ) sprintf(buf, "=%u", link_p->V_UINT );
else if(IS_LONG(link_p)  ) sprintf(buf, "=%ld",link_p->V_LONG );
else if(IS_ULONG(link_p) ) sprintf(buf, "=%lu",link_p->V_ULONG);
else               continue;
```

Disk only. *Page 241, Listing 4.7, lines 586–589 and line 596, and page 399 Listing 5.14, lines 320–323 and line 331. The code in llama.par was not testing correctly for a* `NULL` return value from `ii_ptext()`. The problem has been fixed on the disk, but won't be fixed in the book until the second edition. The fix looks like this:

```
if( yytext = (char *) ii_ptext() )  /* replaces llama.par lines 586-589 */
{                   /* and occs.par, lines 320-323      */
    yylineno       = ii_plineno() ;
    tchar          = yytext[ yyleng = ii_plength() ];
    yytext[yyleng] = '\0' ;
}
else /* no previous token */
{
    yytext = "";
    yyleng = yylineno = 0;
}

if( yylineno )              /* replaces llama.par, line 596 */
    ii_ptext()[ ii_plength() ] = tchar; /* and occs.par, line 331       */
```

Disk only. *The ii_look( ) routine (in Listing 2.7 on page 47) doesn't work in the 8086 large or compact models. The following is ugly, but it works everywhere:*

```
int ii_look( n )
{
    /* Return the nth character of lookahead, EOF if you try to look past
     * end of file, or 0 if you try to look past either end of the buffer.
     * We have to jump through hoops here to satisfy the ANSI restriction
     * that a pointer can not go to the left of an array or more than one
     * cell past the right of an array. If we don't satisfy this restriction,
     * then the code won't work in the 8086 large or compact models. In
     * the small model---or in any machine without a segmented address
     * space, you could do a simple comparison to test for overflow:
     *      uchar *p = Next + n;
     *      if( !(Start_buf <= p && p < End_buf )
     *          overflow
     */

    if( n > (End_buf-Next) )         /* (End_buf-Next) is the # of unread */
        return Eof_read ? EOF : 0 ;  /* chars in the buffer (including    */
                    /* the one pointed to by Next).      */

    /* The current lookahead character is at Next[0].  The last character */
    /* read is at Next[-1]. The --n in the following if statement adjusts */
    /* n so that Next[n] will reference the correct character.        */

```

➡

Listing 5.11. continued...

```
    if( --n < -(Next-Start_buf) )    /* (Next-Start) is the # of buffered */
        return 0;           /* characters that have been read.   */

    return Next[n];
}
```

This page blank.

This page blank.

This page blank.