This example can be easily augmented to handle nested comments, via a global variable that is incremented and decremented in the semantic actions.

PROGRAM LEXICAL ANALYSIS

Use JLex to implement a lexical analyzer for the Tiger language. Appendix A describes, among other things, the lexical tokens of Tiger.

This chapter has left out some of the specifics of how the lexical analyzer should be initialized and how it should communicate with the rest of the compiler. You can learn this from the JLex manual, but the "skeleton" files in the `$TIGER/chap2` directory will also help get you started.

Along with the `Tiger.lex` file you should turn in documentation for the following points:

- how you handle comments;
- how you handle strings;
- error handling;
- end-of-file handling;
- other interesting features of your lexer.

Supporting files are available in `$TIGER/chap2` as follows:

`Parse/sym.java` Class `sym` with definitions of the token-kind constants.
`java_cup/runtime/Symbol.java` This class is provided in the `runtime` directory of the CUP parser generator and is used by the lexical analyzer to report tokens (with associated semantic values) to the parser.
`Parse/Lexer.java` The `Lexer` interface, to be supported by the lexical analyzer.
`ErrorMsg/ErrorMsg.java` The `ErrorMsg` package, useful for producing error messages with file names and line numbers.
`Parse/Main.java` A test scaffold to run your lexer on an input file.
`Parse/Tiger.lex` The beginnings of a lexical analyzer specification.
`makefile` A "makefile" to compile everything.

When reading the *Tiger Language Reference Manual* (Appendix A), pay particular attention to the paragraphs with the headings **Identifiers, Comments, Integer literal,** and **String literal.**

The reserved words of the language are: `while`, `for`, `to`, `break`, `let`, `in`, `end`, `function`, `var`, `type`, `array`, `if`, `then`, `else`, `do`, `of`, `nil`.

The punctuation symbols used in the language are:

```
, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=
```

The string value that you return for a string literal should have all the escape sequences translated into their meanings.

There are no negative integer literals; return two separate tokens for `-32`.

Detect unclosed comments (at end of file) and unclosed strings.

The directory `$TIGER/testcases` contains a few sample Tiger programs.

To get started: Make a directory and copy the contents of `$TIGER/chap2` into it. Make a file `test.tig` containing a short program in the Tiger language. Then type `make`; JLex will run on `Tiger.lex`, producing `Yylex.java`, and then the appropriate Java files will be compiled.

Finally, `java Parse.Main test.tig` will lexically analyze the file using a test scaffold.

FURTHER READING

Lex was the first lexical-analyzer generator based on regular expressions [Lesk 1975]; it is still widely used.

Computing ϵ-closure can be done more efficiently by keeping a queue or stack of states whose edges have not yet been checked for ϵ-transitions [Aho et al. 1986]. Regular expressions can be converted directly to DFAs without going through NFAs [McNaughton and Yamada 1960; Aho et al. 1986].

DFA transition tables can be very large and sparse. If represented as a simple two-dimensional matrix (*states* × *symbols*) they take far too much memory. In practice, tables are compressed; this reduces the amount of memory required, but increases the time required to look up the next state [Aho et al. 1986].

Lexical analyzers, whether automatically generated or handwritten, must manage their input efficiently. Of course, input is buffered, so that a large batch of characters is obtained at once; then the lexer can process one character at a time in the buffer. The lexer must check, for each character, whether the end of the buffer is reached. By putting a *sentinel* – a character that cannot be part of any token – at the end of the buffer, it is possible for the lexer to check for end-of-buffer only once per token, instead of once per character [Aho et al. 1986]. Gray [1988] uses a scheme that requires only one check per line, rather than one per token, but cannot cope with tokens that contain end-of-line characters. Bumbulis and Cowan [1993] check only once around each cycle in the DFA; this reduces the number of checks (from once per