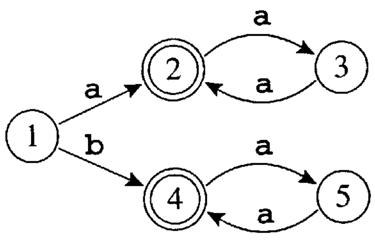
regular expressions that constitute the lexical specification. This is how *rule priority* is implemented.

After the DFA is constructed, the "states" array may be discarded, and the "trans" array is used for lexical analysis.

Applying the DFA construction algorithm to the NFA of Figure 2.7 gives the automaton in Figure 2.8.

This automaton is suboptimal. That is, it is not the smallest one that recognizes the same language. In general, we say that two states s_1 and s_2 are equivalent when the machine starting in s_1 accepts a string σ if and only if starting in s_2 it accepts σ. This is certainly true of the states labeled $\boxed{5,6,7,8,15}$ and $\boxed{6,7,8}$ in Figure 2.8; and of the states labeled $\boxed{10,11,12,13,15}$ and $\boxed{11,12,13}$. In an automaton with two equivalent states s_1 and s_2, we can make all of s_2's incoming edges point to s_1 instead and delete s_2.

How can we find equivalent states? Certainly, s_1 and s_2 are equivalent if they are both final or both non-final and for any symbol c, $\text{trans}[s_1, c] = \text{trans}[s_2, c]$; $\boxed{10,11,12,13,15}$ and $\boxed{11,12,13}$ satisfy this criterion. But this condition is not sufficiently general; consider the automaton



Here, states 2 and 4 are equivalent, but $\text{trans}[2, a] \neq \text{trans}[4, a]$.

After constructing a DFA it is useful to apply an algorithm to minimize it by finding equivalent states; see Exercise 2.6.

2.5 Lex: A LEXICAL ANALYZER GENERATOR

DFA construction is a mechanical task easily performed by computer, so it makes sense to have an automatic *lexical analyzer generator* to translate regular expressions into a DFA.

Lex is a lexical analyzer generator that produces a C program from a *lexical specification*. For each token type in the programming language to be lexically analyzed, the specification contains a regular expression and an *action*.

```
%{
/* C Declarations: */
#include "tokens.h"   /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits  [0-9]+
%%
/* Regular Expressions and Actions: */
if                      {ADJ; return IF;}
[a-z][a-z0-9]*          {ADJ; yylval.sval=String(yytext);
                          return ID;}
{digits}               {ADJ; yylval.ival=atoi(yytext);
                          return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})     {ADJ;
                          yylval.fval=atof(yytext);
                          return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+     {ADJ;}
.                       {ADJ; EM_error("illegal character");}
```

PROGRAM 2.9. Lex specification of the tokens from Figure 2.2.

The action communicates the token type (perhaps along with other information) to the next phase of the compiler.

The output of Lex is a program in C – a lexical analyzer that interprets a DFA using the algorithm described in Section 2.3 and executes the action fragments on each match. The action fragments are just C statements that return token values.

The tokens described in Figure 2.2 are specified in Lex as shown in Program 2.9.

The first part of the specification, between the `%{···%}` braces, contains `includes` and declarations that may be used by the C code in the remainder of the file.

The second part of the specification contains regular-expression abbreviations and state declarations. For example, the declaration `digits [0-9]+` in this section allows the name `{digits}` to stand for a nonempty sequence of digits within regular expressions.

The third part contains regular expressions and actions. The actions are fragments of ordinary C code. Each action must return a value of type `int`, denoting which kind of token has been found.

In the action fragments, several special variables are available. The string matched by the regular expression is `yytext`. The length of the matched string is `yyleng`.

In this particular example, we keep track of the position of each token, measured in characters since the beginning of the file, in the variable `charPos`. The `EM_tokPos` variable of the error message module `errormsg.h` is continually told this position by calls to the macro `ADJ`. The parser will be able to use this information in printing informative syntax error messages.

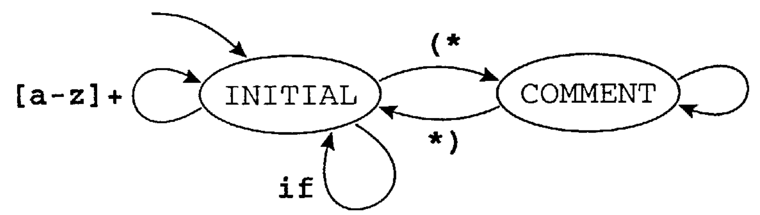
The include file `tokens.h` in this example defines integer constants `IF`, `ID`, `NUM`, and so on; these values are returned by the action fragments to tell what token-type is matched.

Some tokens have *semantic values* associated with them. For example, `ID`'s semantic value is the character string constituting the identifier; `NUM`'s semantic value is an integer; and `IF` has no semantic value (any `IF` is indistinguishable from any other). The values are communicated to the parser through the global variable `yylval`, which is a union of the different types of semantic values. The token-type returned by the lexer tells the parser which variant of the union is valid.

START STATES

Regular expressions are *static* and *declarative*; automata are *dynamic* and *imperative*. That is, you can see the components and structure of a regular expression without having to simulate an algorithm, but to understand an automaton it is often necessary to "execute" it in your mind. Thus, regular expressions are usually more convenient to specify the lexical structure of programming-language tokens.

But sometimes the step-by-step, state-transition model of automata is appropriate. Lex has a mechanism to mix states with regular expressions. One can declare a set of *start states*; each regular expression can be prefixed by the set of start states in which it is valid. The action fragments can explicitly change the start state. In effect, we have a finite automaton whose edges are labeled, not by single symbols, but by regular expressions. This example shows a language with simple identifiers, `if` tokens, and comments delimited by `(*` and `*)` brackets:



Though it is possible to write a single regular expression that matches an entire comment, as comments get more complicated it becomes more difficult, or even impossible if nested comments are allowed.

The Lex specification corresponding to this machine is

```
    :  the usual preamble ...
%Start INITIAL COMMENT
%%
<INITIAL>if       {ADJ; return IF;}
<INITIAL>[a-z]+   {ADJ; yylval.sval=String(yytext); return ID;}
<INITIAL>"(*"     {ADJ; BEGIN COMMENT;}
<INITIAL>.        {ADJ; EM_error("illegal character");}
<COMMENT>"*)"     {ADJ; BEGIN INITIAL;}
<COMMENT>.        {ADJ;}
                  {BEGIN INITIAL; yyless(1);}
```

where INITIAL is the "outside of any comment" state. The last rule is a hack to get Lex into this state. Any regular expression not prefixed by a `<STATE>` operates in all states; this feature is rarely useful.

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 Lex 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 Lex 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:

`tokens.h` Definition of lexical-token constants, and `yylval`.
`errormsg.h, errormsg.c` The error message module, useful for producing error messages with file names and line numbers.
`driver.c` A test scaffold to run your lexer on an input file.
`tiger.lex` The beginnings of a real `tiger.lex` file.
`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`; Lex will run on `tiger.lex`, producing `lex.yy.c`, and then the appropriate C files will be compiled.

Finally, `lextest 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* $\times$ *symbols*) they take far too much mem-