EXPRESSION	MATCHES	EXAMPLE
$'s'$	string s literally	`'\'`
$\backslash c$	character c literally	`\'`
$*$	any string	`*.o`
?	any character	`sort1.?`
$[s]$	any character in s	`sort1.[cso]`

Figure 3.9: Filename expressions used by the shell command `sh`

expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

! **Exercise 3.3.12:** SQL allows a rudimentary form of patterns in which two characters have special meaning: underscore (_) stands for any one character and percent-sign (%) stands for any string of 0 or more characters. In addition, the programmer may define any character, say e, to be the escape character, so an e preceding _, %, or another e gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

3.4 Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

$$\begin{array}{rcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$

Figure 3.10: A grammar for branching statements

Example 3.8: The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

For **relop**, we use the comparison operators of languages like Pascal or SQL, where `=` is "equals" and `<>` is "not equals," because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11. The patterns for *id* and *number* are similar to what we saw in Example 3.7.

$$\begin{array}{rcl}
\mathit{digit} & \rightarrow & [\texttt{0-9}] \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & [\texttt{A-Za-z}] \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}$$

Figure 3.11: Patterns for tokens of Example 3.8

For this language, the lexical analyzer will recognize the keywords `if`, `then`, and `else`, as well as lexemes that match the patterns for *relop*, *id*, and *number*. To simplify matters, we make the common assumption that keywords are also *reserved words*: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

$$\mathit{ws} \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Our goal for the lexical analyzer is summarized in Fig. 3.12. That table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, as discussed in Section 3.1.3, is returned. Note that for the six relational operators, symbolic constants `LT`, `LE`, and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found. The particular operator found will influence the code that is output from the compiler. □

Lexemes	Token Name	Attribute Value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Figure 3.12: Tokens, their patterns, and attribute values

3.4.1 Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams." In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand, but in Section 3.6, we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.

Transition diagrams have a collection of nodes or circles, called *states*. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer (as in the situation of Fig. 3.3).

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer and enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic*, meaning that there is never more than one edge out of a given state with a given symbol among its labels. Starting in Section 3.5, we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always

indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Example 3.9 : Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop**. We begin in state 0, the start state. If we see `<` as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at `<`, `<>`, or `<=`. We therefore go to state 1, and look at the next character. If it is `=`, then we recognize lexeme `<=`, enter state 2, and return the token **relop** with attribute `LE`, the symbolic constant representing this particular comparison operator. If in state 1 the next character is `>`, then instead we have lexeme `<>`, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is `<`, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

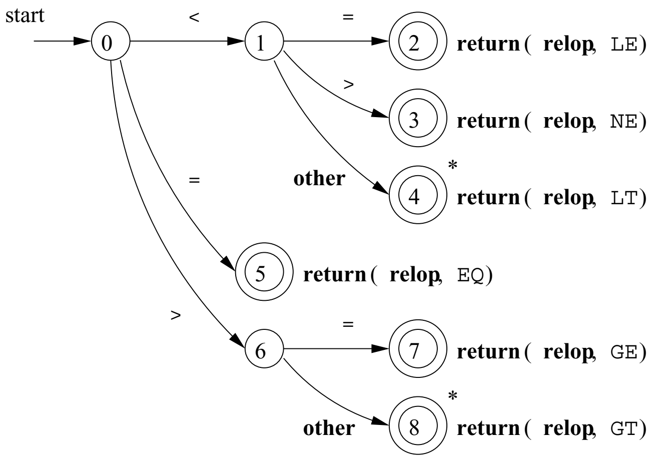

Figure 3.13: Transition diagram for **relop**

On the other hand, if in state 0 the first character we see is `=`, then this one character must be the lexeme. We immediately return that fact from state 5.

The remaining possibility is that the first character is `>`. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is `>=` (if we next see the `=` sign), or just `>` (on any other character). Note that if, in state 0, we see any character besides `<`, `=`, or `>`, we can not possibly be seeing a `relop` lexeme, so this transition diagram will not be used. □

3.4.2 Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like `if` or `then` are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords `if`, `then`, and `else` of our running example.

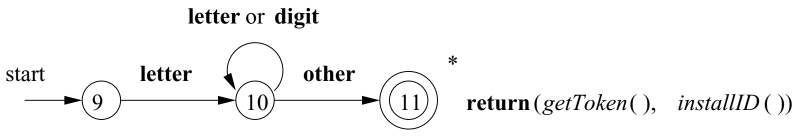

Figure 3.14: A transition diagram for **id**'s and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id**. The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword `then` is shown in Fig. 3.15. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was **id**, with a lexeme like `thenextvalue` that has `then` as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word

tokens are recognized in preference to **id**, when the lexeme matches both patterns. We *do not* use this approach in our example, which is why the states in Fig. 3.15 are unnumbered.

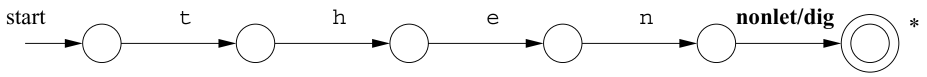

Figure 3.15: Hypothetical transition diagram for the keyword `then`

3.4.3 Completion of the Running Example

The transition diagram for **id**'s that we saw in Fig. 3.14 has a simple structure. Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so. We stay in state 10 as long as the input contains letters and digits. When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found. Since the last character is not part of the identifier, we must retract the input one position, and as discussed in Section 3.4.2, we enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

The transition diagram for token **number** is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit, dot, or `E`, we have seen a number in the form of an integer; `123` is an example. That case is handled by entering state 20, where we return token **number** and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

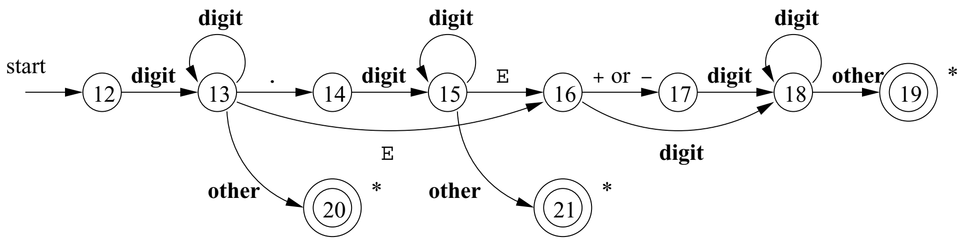

Figure 3.16: A transition diagram for unsigned numbers

If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an `E`, then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but `E` or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by **delim** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.

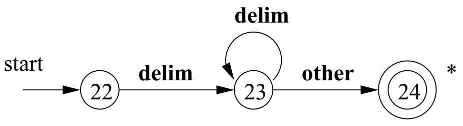

Figure 3.17: A transition diagram for whitespace

Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character. We retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

3.4.4 Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable `state` holding the number of the current state for a transition diagram. A switch based on the value of `state` takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Example 3.10: In Fig. 3.18 we see a sketch of `getRelop()`, a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type `TOKEN`, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case). `getRelop()` first creates a new object `retToken` and initializes its first component to `RELOP`, the symbolic code for token **relop**.

We see the typical behavior of a state in case 0, the case where the current state is 0. A function `nextChar()` obtains the next character from the input and assigns it to local variable c. We then check c for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case. For example, if the next input character is `=`, we go to state 5.

If the next input character is not one that can begin a comparison operator, then a function `fail()` is called. What `fail()` does depends on the global error-recovery strategy of the lexical analyzer. It should reset the `forward` pointer to `lexemeBegin`, in order to allow another transition diagram to be applied to

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

the true beginning of the unprocessed input. It might then change the value of `state` to be the start state for another transition diagram, which will search for another token. Alternatively, if there is no other transition diagram that remains unused, `fail()` could initiate an error-correction phase that will try to repair the input and find a lexeme, as discussed in Section 3.1.4.

We also show the action for state 8 in Fig. 3.18. Because state 8 bears a *, we must retract the input pointer one position (i.e., put c back on the input stream). That task is accomplished by the function `retract()`. Since state 8 represents the recognition of lexeme `>`, we set the second component of the returned object, which we suppose is named `attribute`, to `GT`, the code for this operator. □

To place the simulation of one transition diagram in perspective, let us consider the ways code like Fig. 3.18 could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function `fail()` of Example 3.10 resets the pointer `forward` and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one suggested in Fig. 3.15. We have only to use these before we use the diagram for **id**, in order for the keywords to be reserved words.

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier `thenext` to keyword `then`, or the operator `->` to `-`, for example.

3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.

3.4.5 Exercises for Section 3.4

Exercise 3.4.1: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.2.

Exercise 3.4.2: Provide transition diagrams to recognize the same languages as each of the regular expressions in Exercise 3.3.5.

The following exercises, up to Exercise 3.4.12, introduce the Aho-Corasick algorithm for recognizing a collection of keywords in a text string in time proportional to the length of the text and the sum of the length of the keywords. This algorithm uses a special form of transition diagram called a *trie*. A trie is a tree-structured transition diagram with distinct labels on the edges leading from a node to its children. Leaves of the trie represent recognized keywords.

Knuth, Morris, and Pratt presented an algorithm for recognizing a single keyword $b_1 b_2 \cdots b_n$ in a text string. Here the trie is a transition diagram with $n + 1$ states, 0 through n. State 0 is the initial state, and state n represents acceptance, that is, discovery of the keyword. From each state s from 0 through $n - 1$, there is a transition to state $s + 1$, labeled by symbol b_{s+1}. For example, the trie for the keyword `ababaa` is:

$$\begin{tikzcd}
0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "a"] & 3 \arrow[r, "b"] & 4 \arrow[r, "a"] & 5 \arrow[r, "a"] & \circledcirc 6
\end{tikzcd}$$

In order to process text strings rapidly and search those strings for a keyword, it is useful to define, for keyword $b_1 b_2 \cdots b_n$ and position s in that keyword (corresponding to state s of its trie), a *failure function*, $f(s)$, computed as in

Fig. 3.19. The objective is that $b_1 b_2 \cdots b_{f(s)}$ is the longest proper prefix of $b_1 b_2 \cdots b_s$ that is also a suffix of $b_1 b_2 \cdots b_s$. The reason $f(s)$ is important is that if we are trying to match a text string for $b_1 b_2 \cdots b_n$, and we have matched the first s positions, but we then fail (i.e., the next position of the text string does not hold b_{s+1}), then $f(s)$ is the longest prefix of $b_1 b_2 \cdots b_n$ that could possibly match the text string up to the point we are at. Of course, the next character of the text string must be $b_{f(s)+1}$, or else we still have problems and must consider a yet shorter prefix, which will be $b_{f(f(s))}$.

```
1)  $t = 0$;
2)  $f(1) = 0$;
3)  for ($s = 1$; $s < n$; $s++$) {
4)          while ($t > 0$ && $b_{s+1}$ != $b_{t+1}$) $t = f(t)$;
5)          if ($b_{s+1} == b_{t+1}$) {
6)                  $t = t + 1$;
7)                  $f(s+1) = t$;
            }
8)          else $f(s+1) = 0$;
    }
```

Figure 3.19: Algorithm to compute the failure function for keyword $b_1 b_2 \cdots b_n$

As an example, the failure function for the trie constructed above for `ababaa` is:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

For instance, states 3 and 1 represent prefixes `aba` and `a`, respectively. $f(3) = 1$ because `a` is the longest proper prefix of `aba` that is also a suffix of `aba`. Also, $f(2) = 0$, because the longest proper prefix of `ab` that is also a suffix is the empty string.

Exercise 3.4.3: Construct the failure function for the strings:

a) `abababaab`.

b) `aaaaaa`.

c) `abbaabb`.

! **Exercise 3.4.4:** Prove, by induction on s, that the algorithm of Fig. 3.19 correctly computes the failure function.

!! **Exercise 3.4.5:** Show that the assignment $t = f(t)$ in line (4) of Fig. 3.19 is executed at most n times. Show that therefore, the entire algorithm takes only $O(n)$ time on a keyword of length n.

Having computed the failure function for a keyword $b_1b_2\cdots b_n$, we can scan a string $a_1a_2\cdots a_m$ in time $O(m)$ to tell whether the keyword occurs in the string. The algorithm, shown in Fig. 3.20, slides the keyword along the string, trying to make progress by matching the next character of the keyword with the next character of the string. If it cannot do so after matching s characters, then it "slides" the keyword right $s - f(s)$ positions, so only the first $f(s)$ characters of the keyword are considered matched with the string.

```
1)  $s = 0;$
2)  for ($i = 1; i \leq m; i$++) {
3)          while ($s > 0$ && $a_i$ != $b_{s+1}$) $s = f(s);$
4)          if ($a_i$ == $b_{s+1}$) $s = s + 1;$
5)          if ($s$ == $n$) return "yes";
    }
6)  return "no";
```

Figure 3.20: The KMP algorithm tests whether string $a_1a_2\cdots a_m$ contains a single keyword $b_1b_2\cdots b_n$ as a substring in $O(m+n)$ time

Exercise 3.4.6: Apply Algorithm KMP to test whether keyword `ababaa` is a substring of:

a) `abababaab`.

b) `abababbaa`.

!! **Exercise 3.4.7:** Show that the algorithm of Fig. 3.20 correctly tells whether the keyword is a substring of the given string. *Hint*: proceed by induction on i. Show that for all i, the value of s after line (4) is the length of the longest prefix of the keyword that is a suffix of $a_1a_2\cdots a_i$.

!! **Exercise 3.4.8:** Show that the algorithm of Fig. 3.20 runs in time $O(m+n)$, assuming that function f is already computed and its values stored in an array indexed by s.

Exercise 3.4.9: The *Fibonacci strings* are defined as follows:

1. $s_1 =$ `b`.

2. $s_2 =$ `a`.

3. $s_k = s_{k-1}s_{k-2}$ for $k > 2$.

For example, $s_3 =$ `ab`, $s_4 =$ `aba`, and $s_5 =$ `abaab`.

a) What is the length of s_n?

b) Construct the failure function for s_6.

c) Construct the failure function for s_7.

!! d) Show that the failure function for any s_n can be expressed by $f(1) = f(2) = 0$, and for $2 < j \leq |s_n|$, $f(j)$ is $j - |s_{k-1}|$, where k is the largest integer such that $|s_k| \leq j + 1$.

!! e) In the KMP algorithm, what is the largest number of consecutive applications of the failure function, when we try to determine whether keyword s_k appears in text string s_{k+1}?

Aho and Corasick generalized the KMP algorithm to recognize any of a set of keywords in a text string. In this case, the trie is a true tree, with branching from the root. There is one state for every string that is a prefix (not necessarily proper) of any keyword. The parent of a state corresponding to string $b_1 b_2 \cdots b_k$ is the state that corresponds to $b_1 b_2 \cdots b_{k-1}$. A state is accepting if it corresponds to a complete keyword. For example, Fig. 3.21 shows the trie for the keywords `he`, `she`, `his`, and `hers`.

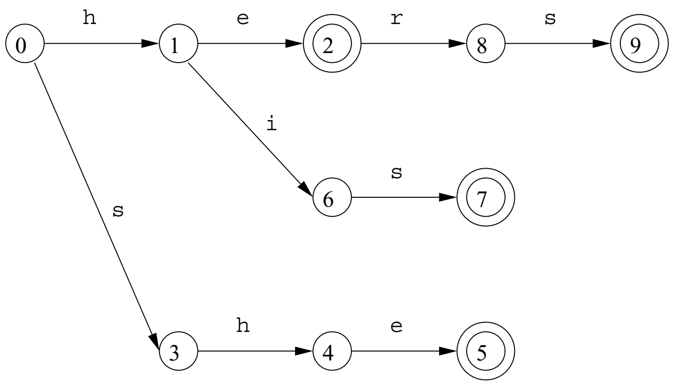

Figure 3.21: Trie for keywords `he`, `she`, `his`, `hers`

The failure function for the general trie is defined as follows. Suppose s is the state that corresponds to string $b_1 b_2 \cdots b_n$. Then $f(s)$ is the state that corresponds to the longest proper suffix of $b_1 b_2 \cdots b_n$ that is also a prefix of *some* keyword. For example, the failure function for the trie of Fig. 3.21 is:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! **Exercise 3.4.10:** Modify the algorithm of Fig. 3.19 to compute the failure function for general tries. *Hint*: The major difference is that we cannot simply test for equality or inequality of b_{s+1} and b_{t+1} in lines (4) and (5) of Fig. 3.19. Rather, from any state there may be several transitions out on several characters, as there are transitions on both `e` and `i` from state 1 in Fig. 3.21. Any of those transitions could lead to a state that represents the longest suffix that is also a prefix.

Exercise 3.4.11: Construct the tries and compute the failure function for the following sets of keywords:

a) `aaa`, `abaaa`, and `ababaaa`.

b) `all`, `fall`, `fatal`, `llama`, and `lame`.

c) `pipe`, `pet`, `item`, `temper`, and `perpetual`.

! **Exercise 3.4.12:** Show that your algorithm from Exercise 3.4.10 still runs in time that is linear in the sum of the lengths of the keywords.

3.5 The Lexical-Analyzer Generator `Lex`

In this section, we introduce a tool called `Lex`, or in a more recent implementation `Flex`, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the `Lex` tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

3.5.1 Use of `Lex`

Figure 3.22 suggests how `Lex` is used. An input file, which we call `lex.l`, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`. The latter file is compiled by the C compiler into a file called `a.out`, as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as `a.out` in Fig. 3.22, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable `yylval`,[2] which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

[2]Incidentally, the `yy` that appears in `yylval` and `lex.yy.c` refers to the `Yacc` parser-generator, which we shall describe in Section 4.9, and which is commonly used in conjunction with `Lex`.