Lexical Analysis (Scanning)

- Goal of scanning is to turn the input stream of characters into a stream of tokens (lexical units of the language)
- Scanning usually removes white space, comments
- A scanner is essentially a finite state machine, also called a deterministic finite automaton (DFA)
- Equivalent to regular expressions

Some terminology

- An alphabet is a finite set of symbols (e.g., ASCII characters)
- A string is a sequence of symbols from the alphabet
- The empty string has length zero, denoted $\varepsilon$
- A language is a set of strings (may be infinite)
- Set operations can be performed
  - Union of two languages is a language
  - Concatenation of two languages is language consisting of all strings formed by appending each string in second set to each string in first set
  - Kleene closure of a language is all strings formed by concatenation of strings from the set (also called the star of the language)
Formal definition of DFA

1) An alphabet
2) A finite set of States
3) A start state
4) One or more accepting or final states
5) A transition function:
   \[ \text{States} \times \text{InputSymbols} \rightarrow \text{States} \]

A DFA defines a language (accepts strings)

DFA as Transition Table

<table>
<thead>
<tr>
<th>states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>accept</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>accept</td>
<td>error</td>
<td>accept</td>
<td>accept</td>
</tr>
<tr>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>

input
Diagram view of the same DFA

Abbreviated view of the same DFA

collapse transitions with single label
Yet another view of the same DFA

Omit error transitions

Same DFA with stopping condition, action

- Brackets indicate input that is checked but not consumed
- "Maximal munch" – match longest substring before stopping

printf("OK");
Typical Token categories

- Reserved words (keywords):
  - if while do ...
- Identifiers:
  - interest x23 __z_over
- Literals (constants):
  - 42 3.14159 “Hello”
- Special symbols:
  - + += ; , ==
- White space: blanks, tabs, comments, newlines, other control characters

TINY Tokens

<table>
<thead>
<tr>
<th>Reserved words</th>
<th>Special symbols</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>+</td>
<td>number</td>
</tr>
<tr>
<td>then</td>
<td>−</td>
<td>(1 or more</td>
</tr>
<tr>
<td>else</td>
<td>*</td>
<td>digits)</td>
</tr>
<tr>
<td>end</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>repeat</td>
<td>=</td>
<td>identifier</td>
</tr>
<tr>
<td>until</td>
<td>&lt;</td>
<td>(1 or more</td>
</tr>
<tr>
<td>read</td>
<td>)</td>
<td>letters)</td>
</tr>
<tr>
<td>write</td>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>
A DFA for Numbers in TINY

Turning a DFA into an Algorithm

- States are an enum, variable for current
- Main loop
- Switch statement on the state in the loop
- Each state has a nested switch (or if) statement on the input symbol
Sample code for a TINY number

```c
state = START; input = next();
while (state != DONE && state != ERR) {
    switch (state) {
        case START: if (isdigit(input)) {
            input = next(); state = INNUM;
            } else state = ERR; break;
        case INNUM: if (!isdigit(input))
            state = DONE;
            else input = next(); break;
        default: break;
    }
    if (state == DONE) return NUM;
    else return ERROR;
}
```

Alternative code

```c
state = START; input = next();
while (TRUE) {
    switch (state) {
        case START: if (isdigit(input)) {
            input = next(); state = INNUM;
            } else return ERROR; break;
        case INNUM: if (!isdigit(input))
            return NUM;
            else input = next(); break;
        default: break;
    }
```
Table-driven code

```c
state = START; input = next();
while (!accept[state] && state != ERR){
    newstate = t[state][input];
    if (adv[state][input])
        input = next();
    state = newstate;
}
...  
- t is array of transition states
- accept is array of accepting states
```

TINY DFA:

![DFA Diagram]

- INNUM
- INASSIGN
- DONE
- START
- INCOMMENT
- INID
- digit
- letter
- other
- whitespace
Handling reserved words

- Recognize them as identifiers first
- Then look up in a table
- Use efficient search techniques (binary search, hash tables) for speed

Token information

- Token type (an enum)
- Text of token (the "lexeme")
- May perform conversions
  - String to number
  - Keyword to value (e.g., true, false)
- May start building symbol table
- Line number
  - Maybe character position
  - Maybe entire text of line
- File name (if dealing with multiple files)
Turning token descriptions into a DFA

- Ad hoc methods are error-prone
- Maintaining determinism can be tricky
- Even the English token descriptions themselves might be ambiguous or subject to interpretation

Example
- A comment in C is any sequence of characters between the delimiters "/*" and "*/"
- The following DFA is wrong!
Example, continued
Here is the correct DFA:

Other questions:
• What if EOF is encountered inside?
• What if a comment is encountered in a string?
• What if a string is encountered in a comment?
• Do comments count as white space or not?

Another Example
The following automaton is nondeterministic:

return PLUS_EQ
return INC
return PLUS
Another Example, Continued

It must be replaced by this:

A Better Approach:

- **Use regular expressions to define the tokens.**
- Use well-known (and correct) mathematical algorithms to convert the tokens into a DFA.
- Even better: automate the entire process from regular expressions to code:
  - lex (unix) - 1975 (Mike Lesk)
  - flex (usenix) - 1985 (Van Jacobson, Vern Paxson)
  - Jflex – 1998 (Gerwin Klein)
Implementing a Scanner - Regular Expressions and Lex

A regular expression is an expression that matches strings (matched set is the “language” of the RE)

A regular expression is built from basic expressions and operations (individual symbols) and the operations choice (|), concatenation and repetition (*)

Definition of Regular Expression

A basic regular expression is:
- A single symbol
- The empty string ε
- The null expression ∅

Operators on REs produce REs:
- Choice (union) – operator |
- Concatenation – no operator symbol
- Repetition (zero or more occurrences) – operator *

- May also use parentheses for grouping
Regular Expression Examples

\[ \text{a } (\text{b } | \text{c}) \star \]
- strings beginning with a, followed by any number of b's and c's

\[ \text{ab } | \text{c} \star \]
- the string ab, or the strings ε, c, cc, ccc, ....

\[ (\text{b } | \epsilon) \ (\text{a } | \text{ab}) \star \]
- strings of a's and b's with no two consecutive b's.

Common Extensions of RE ops

- +  one or more repetitions (\(r^+\) same as \(rr^*\))
- [ ]  sets (\([abcd] = a | b | c | d = [a-d]\))
- .  any character
- \(\text{\textbackslash}\)  “escape” an operator or metasymbol
- ^  negate a set (\([^abc] = \text{any char except } a, b, \text{ or } c\))
- ?  optional - zero or one (\(b? (ab?) \star\))
More complex examples

  or:
  
  \(\text{nat} = [0-9]+\)
  
  \(\text{signedNat} = [-+]? \text{nat}\)
  
  \(\text{number} = \text{signedNat}(\. \text{nat})?\)
  
  \(([Ee] \text{signedNat})?\)

- \(\text{circle6} /\text{circle6} /\text{circle6} (\text{circle6}[^/*]/*)*\text{circle6}\)

---

Lex: A Scanner Generator

Regular Expression

\[\text{Regular Expression} \rightarrow \text{Thompson’s Construction} \rightarrow \text{Nondeterministic Finite Automaton}\]

\[\rightarrow \text{“Subset” Construction} \rightarrow \text{Deterministic Finite Automaton}\]

\[\rightarrow \text{Table-driven Scanner}\]