Scanner

- maps characters into *tokens* – the basic unit of syntax
  
  \[ x = x + y; \]
  
  becomes
  
  \[ <\text{id}, x> = <\text{id}, x> + <\text{id}, y> ; \]

- character string value for a *token* is a *lexeme*

- typical tokens: *number, id, +, -, *, /, do, end*

- eliminates white space (*tabs, blanks, comments*)

- a key issue is speed
  
  \[ \Rightarrow \text{use specialized recognizer (as opposed to lex)} \]
Specifying patterns

A scanner must recognize the units of syntax
Some parts are easy:

white space
  \( \langle \text{ws} \rangle \) ::= \( \langle \text{ws} \rangle \) \ ' ' |
  \( \langle \text{ws} \rangle \) \ 't'
  ' ' |
  '\t'

keywords and operators
  specified as literal patterns: do, end

comments
  opening and closing delimiters: /* ⋮ */
Specifying patterns

A scanner must recognize the units of syntax
Other parts are much harder:

*identifiers*
  alphabetic followed by \( k \) alphanumerics (\(-, $, \&\), \ldots\)

*numbers*
  integers: 0 or digit from 1-9 followed by digits from 0-9
  decimals: integer \( \cdot \) digits from 0-9
  reals: (integer or decimal) \( \cdot E \) (+ or -) digits from 0-9
  complex: \( '(\cdot \text{real} \cdot, \cdot \text{real} \cdot)' \)

We need a powerful notation to specify these patterns
### Operations on languages

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>union of</strong> $L$ and $M$</td>
<td>$L \cup M = { s \mid s \in L \text{ or } s \in M }$</td>
</tr>
<tr>
<td><strong>written</strong> $L \cup M$</td>
<td></td>
</tr>
<tr>
<td><strong>concatenation of</strong> $L$ and $M$</td>
<td>$LM = { st \mid s \in L \text{ and } t \in M }$</td>
</tr>
<tr>
<td><strong>written</strong> $LM$</td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of</strong> $L$</td>
<td>$L^* = \bigcup_{i=0}^{\infty} L^i$</td>
</tr>
<tr>
<td><strong>written</strong> $L^*$</td>
<td></td>
</tr>
<tr>
<td><strong>positive closure of</strong> $L$</td>
<td>$L^+ = \bigcup_{i=1}^{\infty} L^i$</td>
</tr>
<tr>
<td><strong>written</strong> $L^+$</td>
<td></td>
</tr>
</tbody>
</table>
Regular expressions

Patterns are often specified as *regular languages*

Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*

**Regular expressions (over an alphabet Σ):**

1. ε is a RE denoting the set {ε}
2. if \( a \in \Sigma \), then \( a \) is a RE denoting \( \{a\} \)
3. if \( r \) and \( s \) are REs, denoting \( L(r) \) and \( L(s) \), then:
   
   \( (r) \) is a RE denoting \( L(r) \)
   
   \( (r) \mid (s) \) is a RE denoting \( L(r) \cup L(s) \)
   
   \( (r)(s) \) is a RE denoting \( L(r)L(s) \)
   
   \( (r)^* \) is a RE denoting \( L(r)^* \)

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.
Examples

identifier
\[
\text{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)
\]
\[
\text{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)
\]
\[
\text{id} \rightarrow \text{letter} \ (\text{letter} \mid \text{digit})^*
\]

numbers
\[
\text{integer} \rightarrow (+ \mid − \mid \varepsilon) \ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9) \ \text{digit}^*)
\]
\[
\text{decimal} \rightarrow \text{integer} \ (\text{digit}^*)
\]
\[
\text{real} \rightarrow (\text{integer} \mid \text{decimal}) \ \varepsilon \ (\pm \mid −) \ \text{digit}^*
\]
\[
\text{complex} \rightarrow '(' \ \text{real} \ , \ \text{real} \ ')'
\]

Numbers can get much more complicated

Most programming language tokens can be described with REs

We can use REs to build scanners automatically
## Algebraic properties of REs

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r</td>
<td>s = s</td>
</tr>
<tr>
<td>$r</td>
<td>(s</td>
</tr>
<tr>
<td>$(rs)t = r(st)$</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>$r(s</td>
<td>t) = rs</td>
</tr>
<tr>
<td>$(s</td>
<td>t)r = sr</td>
</tr>
<tr>
<td>$\varepsilon r = r$</td>
<td>$\varepsilon$ is the identity for concatenation</td>
</tr>
<tr>
<td>$r\varepsilon = r$</td>
<td></td>
</tr>
<tr>
<td>$r^* = (r</td>
<td>\varepsilon)^*$</td>
</tr>
<tr>
<td>$r^{**} = r^*$</td>
<td>$*$ is idempotent</td>
</tr>
</tbody>
</table>
Examples

Let $\Sigma = \{a, b\}$

1. $a|b$ denotes $\{a, b\}$

2. $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
   i.e., $(a|b)(a|b) = aa|ab|ba|bb$

3. $a^*$ denotes $\{\varepsilon, a, aa, aaa, \ldots\}$

4. $(a|b)^*$ denotes the set of all strings of $a$'s and $b$'s (including $\varepsilon$)
   i.e., $(a|b)^* = (a^*b^*)^*$

5. $a|a^*b$ denotes $\{a, b, ab, aab, aaab, aaaaab, \ldots\}$
Recognizers

From a regular expression we can construct a
deterministic finite automaton (DFA)

Recognizer for identifier:

\[
\text{identifier} \\
\text{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z) \\
\text{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \\
\text{id} \rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^*
\]
Code for the recognizer

```c
char ← next_char();
state ← 0; /* code for state 0 */
done ← false;
token_value ← "" /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case 1: /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            break;
        case 2: /* accept state */
            token_type = identifier;
            done = true;
            break;
        case 3: /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;
```
Tables for the recognizer

Two tables control the recognizer

<table>
<thead>
<tr>
<th>char_class:</th>
<th>$a-z$</th>
<th>$A-Z$</th>
<th>$0-9$</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>digit</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To change languages, we can just change tables
Automatic construction

Scanner generators automatically construct code from RE-like descriptions

- construct a DFA
- use state minimization techniques
- emit code for the scanner
  (table driven or direct code)

A key issue in automation is an interface to the parser

**lex** is a scanner generator supplied with UNIX

- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)
Grammars for regular languages

Can we place a restriction on the form of a grammar to ensure that it describes a regular language?

Provable fact:

For any RE $r$, $\exists$ a grammar $g$ such that $L(r) = L(g)$

Grammars that generate regular sets are called regular grammars:

They have productions in one of 2 forms:

1. $A \rightarrow aA$
2. $A \rightarrow a$

where $A$ is any non-terminal and $a$ is any terminal symbol

These are also called type 3 grammars (Chomsky)
More regular languages

Example: the set of strings containing an even number of zeros and an even number of ones

\[
\begin{align*}
S_0 & \xrightarrow{0} S_0 & \xrightarrow{1} S_1 \\
S_1 & \xrightarrow{1} S_1 & \xrightarrow{0} S_2 \\
S_2 & \xrightarrow{0} S_2 & \xrightarrow{1} S_3 \\
S_3 & \xrightarrow{0} S_3 & \xrightarrow{0} S_0
\end{align*}
\]

The RE is \((00 | 11)^*(01 | 10)(00 | 11)^*(01 | 10)(00 | 11)^*)^*\)
More regular expressions

What about the RE \((a \mid b)^*abb\) ?

State \(s_0\) has multiple transitions on \(a\)!
\[ \Rightarrow \text{nondeterministic finite automaton} \]

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>({s_0, s_1})</td>
<td>({s_0})</td>
</tr>
<tr>
<td>(s_1)</td>
<td>(_)</td>
<td>({s_2})</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(_)</td>
<td>({s_3})</td>
</tr>
</tbody>
</table>
Finite automata

A non-deterministic finite automaton (NFA) consists of:

1. a set of states \( S = \{s_0, \ldots, s_n\} \)
2. a set of input symbols \( \Sigma \) (the alphabet)
3. a transition function \( \text{move} \) mapping state-symbol pairs to sets of states
4. a distinguished start state \( s_0 \)
5. a set of distinguished accepting or final states \( F \)

A Deterministic Finite Automaton (DFA) is a special case of an NFA:

1. no state has a \( \varepsilon \)-transition, and
2. for each state \( s \) and input symbol \( a \), there is at most one edge labelled \( a \) leaving \( s \)

A DFA accepts \( x \) iff. \( \exists \) a unique path through the transition graph from \( s_0 \) to a final state such that the edges spell \( x \).
DFAs and NFAs are equivalent

1. DFAs are clearly a subset of NFAs

2. Any NFA can be converted into a DFA, by simulating sets of simultaneous states:
   - each DFA state corresponds to a set of NFA states
   - possible exponential blowup
NFA to DFA using the subset construction: example 1

The given NFA transitions are:

- From $s_0$, on $a$ go to $s_1$.
- From $s_1$, on $b$ go to $s_2$.
- From $s_2$, on $b$ go to $s_3$.

Using the subset construction, the DFA transitions are:

- From $\{s_0\}$, on $a$ go to $\{s_0, s_1\}$.
- From $\{s_0, s_1\}$, on $b$ go to $\{s_0, s_2\}$.
- From $\{s_0, s_2\}$, on $b$ go to $\{s_0, s_3\}$.
- From $\{s_0\}$, on $b$ go to $\{s_0\}$.

The DFA is as follows:

- States: $\{s_0\}, \{s_0, s_1\}, \{s_0, s_2\}, \{s_0, s_3\}$.
- Transitions: $a$: from $\{s_0\}$ to $\{s_0, s_1\}$; from $\{s_0, s_1\}$ to $\{s_0, s_2\}$; from $\{s_0, s_2\}$ to $\{s_0, s_3\}$.
- Transitions: $b$: from $\{s_0\}$ to $\{s_0\}$; from $\{s_0, s_1\}$ to $\{s_0, s_2\}$; from $\{s_0, s_2\}$ to $\{s_0, s_3\}$.
Constructing a DFA from a regular expression

RE → NFA w/ε moves
  build NFA for each term
  connect them with ε moves

NFA w/ε moves to DFA
  construct the simulation
  the “subset” construction

DFA → minimized DFA
  merge compatible states

DFA → RE
  construct $R^k_{ij} = R^k_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$
RE to NFA

\[ N(\varepsilon) \]

\[ N(a) \]

\[ N(A|B) \]

\[ N(AB) \]

\[ N(A^*) \]
RE to NFA: example

\[ a | b \]

\[ (a | b)^* \]

\[ abb \]
NFA to DFA: the subset construction

Input: NFA $N$
Output: A DFA $D$ with states $Dstates$ and transitions $Dtrans$ such that $L(D) = L(N)$
Method: Let $s$ be a state in $N$ and $T$ be a set of states, and using the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$-closure($s$)</td>
<td>set of NFA states reachable from NFA state $s$ on $\varepsilon$-transitions alone</td>
</tr>
<tr>
<td>$\varepsilon$-closure($T$)</td>
<td>set of NFA states reachable from some NFA state $s$ in $T$ on $\varepsilon$-transitions alone</td>
</tr>
<tr>
<td>move($T, a$)</td>
<td>set of NFA states to which there is a transition on input symbol $a$ from some NFA state $s$ in $T$</td>
</tr>
</tbody>
</table>

add state $T = \varepsilon$-closure($s_0$) unmarked to $Dstates$

while $\exists$ unmarked state $T$ in $Dstates$

mark $T$

for each input symbol $a$

$U = \varepsilon$-closure(move($T, a$))

if $U \notin Dstates$ then add $U$ to $Dstates$ unmarked

$Dtrans[T, a] = U$

endfor

endwhile

$\varepsilon$-closure($s_0$) is the start state of $D$
A state of $D$ is final if it contains at least one final state in $N$
NFA to DFA using subset construction: example 2

\[
\begin{align*}
A &= \{0, 1, 2, 4, 7\} & D &= \{1, 2, 4, 5, 6, 7, 9\} \\
B &= \{1, 2, 3, 4, 6, 7, 8\} & E &= \{1, 2, 4, 5, 6, 7, 10\} \\
C &= \{1, 2, 4, 5, 6, 7\}
\end{align*}
\]
Limits of regular languages

Not all languages are regular

One cannot construct DFAs to recognize these languages:

- \( L = \{p^k q^k\} \)
- \( L = \{wcw^r \mid w \in \Sigma^*\} \)

*Note: neither of these is a regular expression! (DFAs cannot count!)*

But, this is a little subtle. One can construct DFAs for:

- alternating 0’s and 1’s
  \((\varepsilon \mid 1)(01)^*(\varepsilon \mid 0)\)

- sets of pairs of 0’s and 1’s
  \((01 \mid 10)^+\)
Ramification - Internet Protocol

How does your browser establish a connection with a web server?

- The client sends a SYN message to the server.
- In response, the server replies with a SYN-ACK.
- Finally the client sends an ACK back to the server.

This is done through two DFAs in the client and server, respectively.
### Ramification - Intrusion Detection

<table>
<thead>
<tr>
<th>Code</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FILE * f;</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>f=fopen(&quot;demo&quot;, &quot;r&quot;);</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>strcpy(...); //vulnerability</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>if (!f)</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>printf(&quot;Fail to open\n&quot;);</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>else</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>fgets(f, buf);</code></td>
<td><code>------&gt;</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>------&gt;</code></td>
</tr>
</tbody>
</table>

A DFA will be exercised simultaneously with the program on the OS side to detect intrusion.