A scanner must recognize various parts of the language's syntax.

Input is separated into tokens based on lexical analysis.

\[ x = x + y; \]

becomes

\[ <\text{id}, x> = <\text{id}, x> + <\text{id}, y> ; \]

Specifying patterns

A scanner must recognize various parts of the language’s syntax.

Some parts are easy:

white space

some combination of \(<\text{b}>\) and \text{tab}

keywords and operators

specified as literal patterns — do, end

comments

opening and closing delimiters — /* ... */
Specifying patterns

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 “.” digits from 0-9

We need a powerful notation to specify these patterns.

---

Regular expressions

Regular expressions represent languages.
Languages are sets of strings.
Operations include *Kleene closure*, *concatenation*, and *union*.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (a) )</td>
<td>{ &quot;a&quot; }</td>
</tr>
<tr>
<td>( (a) \mid (b) )</td>
<td>{ &quot;a&quot;, &quot;b&quot; }</td>
</tr>
<tr>
<td>( (a)(b) )</td>
<td>{ &quot;ab&quot; }</td>
</tr>
<tr>
<td>( (a)^* )</td>
<td>{ &quot;&quot;, &quot;a&quot;, &quot;aa&quot;, ... }</td>
</tr>
<tr>
<td>( (a)^+ )</td>
<td>{ &quot;a&quot;, &quot;aa&quot;, ... }</td>
</tr>
</tbody>
</table>

We assume *closure*, *concatenation*, *union* as the order of precedence.

\[
ab \mid cd^* = (ab) \mid (c(d^*)) = \{ "ab", "c", "cd", "cdd", ... \}
\]

\[
a(bc)^* = \{ "a", "abc", "abcbc", ... \}
\]
Recognizers

From a regular expression, we can construct a deterministic finite automaton (dfa).

Recognizer for identifier:

letter
digit

S0

letter
S1

S2

digit
other

accept

S3

error

identifier

letter → (a | b | c | ... | z | A | B | C | ... | Z)
digit → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
id → letter (letter | digit)*

Code for the recognizer

char ← next_char(); done ← false;
state ← S0; /* code for S0 */
token_value ← "" /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class, state];
    switch(state) {
        case S1: /* building an id */
            token_value ← token_value + char;
            char ← next_char(); break;
        case S2: /* accept state */
            token_type ← identifier;
            done ← true; break;
        case S3: /* 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>next_state:</th>
<th>class</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>S1</td>
<td>S1</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>digit</td>
<td>S3</td>
<td>S1</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>other</td>
<td>S3</td>
<td>S2</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

To change languages, we can just change tables.

Improved efficiency

Table driven implementation is slow relative to direct code. Each state transition involves:

1. classifying the input character
2. finding the next state
3. an assignment to the state variable
4. a branch
5. a trip through the case statement logic

We can do better by “encoding” the state table in the scanner code.

1. classify the input character
2. test character class locally
3. branch directly to next state

This takes many fewer instructions per cycle.
Faster scanning

token_type ← error;
char ← next_char();
class ← char_class[char];
if (class != letter)
    return token_type;

S1: token_value ← char;
    char ← next_char();
    class ← char_class[char];
    if (class == other) goto S3;

S2: token_value ← token_value + char;
    char ← next_char();
    class ← char_class[char];
    if (class != other) goto S2;

S3: token_type = identifier;
    return token_type;

Nondeterministic finite automata

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

\[
\begin{array}{c}
\text{Start} \\
\varepsilon
\end{array} \quad
\begin{array}{c}
S1 \\
\text{a|b}
\end{array} \quad
\begin{array}{c}
S2 \\
a
\end{array} \quad
\begin{array}{c}
S3 \\
b
\end{array} \quad
\begin{array}{c}
S4 \\
b
\end{array}
\]

State Start has \(\varepsilon\) transition to S1.
State S1 has multiple transitions on a!

\(\Rightarrow\) nondeterministic finite automaton (nfa)

Different definition for accept
A \textit{nfa accepts} \(x\) if and only if there is some path through the transition graph from the start state to an accepting state such that the labels along the edges spell \(x\).
What is the relationship between a NFA and a DFA?

**DFA is a special case of NFA**

1. no ε transitions
2. single-valued transition function

**DFA can be simulated on an NFA**

- obviously

**NFA can be simulated on a DFA**

- simulate sets of simultaneous states
- possible exponential blowup

---

**Constructing a DFA from a regular expression**

*regular expression (RE) → NFA*

*build NFA for each term, connect them with ε moves*

*NFA → DFA*

*construct the simulation, using the “subset” construction*

*DFA → regular expression*

*construct \( R_{ij}^k = R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1} \)
Converting regular expressions to nfas

Build two-state automaton for atomic regular expression a, with a as the edge.

Compose automata as follows:

- concatenate

- union

- Kleene closure

Subset construction algorithm

Input: nfa N
Output: dfa D with Dstates and Dtran that accepts same language
Method: let s be a state in nfa and T a set of states, using the following definitions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$-closure(s)</td>
<td>Set of nfa states reachable from nfa state s on $\epsilon$-transitions alone.</td>
</tr>
<tr>
<td>$\epsilon$-closure(T)</td>
<td>Set of nfa states reachable from some nfa state s in T on $\epsilon$-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>
Subset construction (cont)

state $\text{Start} = \epsilon$-closure$(s_0)$
add $\text{Start}$ unmarked to $D\text{states}$
while $\exists$ an unmarked state $T$ in $D\text{states}$
mark $T$
for each input symbol $a$ do
$U = \epsilon$-closure$(\text{move}(T,a))$
if $U$ is not in $D\text{states}$ then
add $U$ to $D\text{states}$ unmarked
$D\text{tran}[T,a] = U$
endfor
endwhile

Each state in $D$ corresponds to a set of states in $N$.
Up to $2^{|N|}$ possible states in $D$.
$\epsilon$-closure$(s_0)$ is the start state of $D$.
A state is an accepting state in $D$, if one or more of the states it represents in $N$ is accepting.

Example subset construction

nfa for $a^*b$

\[
\begin{array}{c}
\text{MOVE}(, ) = \{ \} \\
\text{MOVE}(, ) = \{ \} \\
\text{MOVE}(, ) = \{ \} \\
\text{MOVE}(, ) = \{ \} \\
\text{MOVE}(, ) = \{ \} \\
\end{array}
\]

\[
\begin{array}{c}
\text{e-closure( )} = \{ \} \\
\text{e-closure( )} = \{ \} \\
\text{e-closure( )} = \{ \} \\
\text{e-closure( )} = \{ \} \\
\text{e-closure( )} = \{ \} \\
\end{array}
\]
Building minimum-state DFAs

Important theoretical result

Every regular language is recognized by a minimum-state DFA that is unique up to state names.

Look for states that can be distinguished from each other (i.e., end up in accepting/nonaccepting state for identical input).

DFA state minimization algorithm

- construct initial partition of states into accepting and non-accepting states
- successively refine partition by splitting a group $G$ into smaller groups if states in $G$ have transitions to different groups
  (two states $x$, $y$ are in same group iff for all input symbols $a$, $x$ and $y$ have transitions to same group)
- update transition edges, remove dead states

Theorem 3.10, pages 67-71 in Hopcroft and Ullman’s book Introduction to Automata Theory, Languages, and Computation

Example minimal DFA construction

DFA for $a^*b$ from NFA

<table>
<thead>
<tr>
<th>Initial partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>accepting = {}</td>
</tr>
<tr>
<td>non-accepting = {}</td>
</tr>
</tbody>
</table>

Split groups

<table>
<thead>
<tr>
<th>(state, input)</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(S, a)$</td>
<td></td>
</tr>
<tr>
<td>$(T, a)$</td>
<td></td>
</tr>
<tr>
<td>$(S, b)$</td>
<td></td>
</tr>
<tr>
<td>$(T, b)$</td>
<td></td>
</tr>
</tbody>
</table>

Minimal DFA
Issues

Complexity Tradeoffs

For regular expression \( r \) and input \( x \)

<table>
<thead>
<tr>
<th></th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( nfa )</td>
<td>( O(</td>
<td>r</td>
</tr>
<tr>
<td>( dfa )</td>
<td>( O(2^{</td>
<td>r</td>
</tr>
</tbody>
</table>

Other approaches

- generate \( dfa \) directly from regular expression
- two stack simulation of \( nfa \)
- "lazy" construction of \( dfa \)

So what is hard?

Language features that can cause problems:

reserved words

- PL/I had no reserved words
- \( \text{if then then then = else;} \)
- \( \text{else else = then;} \)

significant blanks

- FORTRAN and Algol68 ignore blanks
- \( \text{do 10 i = 1,25} \)
- \( \text{do 10 i = 1.25} \)

string constants

- special characters in strings
- newline, tab, quote, comment delimiter

These problems can be swept under the rug (avoided) by intelligent language design.
Lexical errors

What is a lexical error?

- 1234G6
- illegal character

What should the scanner do?

- report the error
- try to correct it?

Error correction techniques

- minimum distance corrections
- hard token recovery
- skip until match

How bad can it get?

```
1 INTEGERFUNCTIONA
2 PARAMETER(A=6,B=2)
3 IMPLICIT CHARACTER*(A-B)(A-B)
4 INTEGER FORMAT(10),IF(10),DO9E1
5  100 FORMAT(4H)=(3)
6  200 FORMAT(4 )=(3)
7   DO9E1=1
8   DO9E1=1,2
9       IF(X)=1
10      IF(X)H=1
11      IF(X)300,200
12  300 CONTINUE
13 END
C this is a comment
$ FILE(1)
14 END
```
Scanners

- break up input into tokens
- catch lexical errors
- difficulty affected by language design

Issues

- input buffering
- lookahead
- error recovery

Scanner generators

- tokens specified by regular expressions
- construct DFA to recognize language
- highly efficient in practice