Scanner

source code → scanner → tokens → parser → il → errors

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 \(< \backslash b >\) and \( \text{tab} \)
keywords and operators
  specified as literal patterns — do, end
comments
  opening and closing delimiters — \(/ * \ldots */\)

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>{ “a” }</td>
</tr>
<tr>
<td>( a \mid (b) )</td>
<td>{ “a”, “b” }</td>
</tr>
<tr>
<td>( a \mid b )</td>
<td>{ “ab” }</td>
</tr>
<tr>
<td>( a^* )</td>
<td>{ “”, “a”, “aa”, … }</td>
</tr>
<tr>
<td>( a^+ )</td>
<td>{ “a”, “aa”, … }</td>
</tr>
</tbody>
</table>

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

\[ ab \mid \alpha^* = (ab) \mid (\alpha(\alpha^*)) \]

\[ = \{ “ab”, “c”, “\alpha\alpha”, “\alpha\alpha\alpha”, … \} \]

\[ a(bc)^* = \{ “a”, “abc”, “a\alpha\beta\gamma”, … \} \]
Recognizers

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

**Recognizer for identifier:**

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>letter</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>digit</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>other</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>letter</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>digit</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>other</td>
<td>accept</td>
</tr>
<tr>
<td>S3</td>
<td>error</td>
<td>S3</td>
</tr>
</tbody>
</table>

**identifier**

\[
\text{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)
\]

\[
\text{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)
\]

\[
\text{id} \rightarrow \text{letter} (\text{letter} \mid \text{digit})^*
\]

**Tables for the recognizer**

Two tables control the recognizer.

<table>
<thead>
<tr>
<th>char_class</th>
<th>value</th>
<th>letter</th>
<th>letter</th>
<th>digit</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-z</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>digit</td>
<td>S3</td>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>S3</td>
<td>S2</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

\[
\text{token_type} \leftarrow \text{error}; \\
\text{char} \leftarrow \text{next_char}(); \\
\text{class} \leftarrow \text{char_class}[	ext{char}]; \\
\text{if} (\text{class} \neq \text{letter}) \\
\quad \text{return token_type}; \\
\]

S1: token_value \leftarrow char; \\
char \leftarrow next_char(); \\
class \leftarrow char_class[char]; \\
if (class \neq \text{other}) goto S3;

S2: token_value \leftarrow token_value + char; \\
char \leftarrow next_char(); \\
class \leftarrow char_class[char]; \\
if (class \neq \text{other}) goto S2;

S3: token_type = identifier; \\
return token_type;

Nondeterministic finite automata

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

\[
\begin{array}{c}
\text{Start} \\
\epsilon \\
\rightarrow \text{S1} \\
\rightarrow \text{S2} \\
\rightarrow \text{S3} \\
\rightarrow \text{S4}
\end{array}
\]

State Start has \(\epsilon\) transition to S1.
State S1 has multiple transitions on a!
⇒ nondeterministic finite automaton (nfa)

Different definition for accept
A 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.

**Constructing a dfa from a regular expression**

regular expression \((RE) \rightarrow nfa\)
build nfa for each term, connect them with \(\epsilon\) moves
nfa\(\rightarrow\) dfa
construct the simulation, using the “subset” construction
dfa\(\rightarrow\) regular expression
construct \(R_{ij}^k = R_{ik}^{k-1} (R_{kj}^{k-1}) R_{ij}^{k-1}\)
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 (cont)

state $Start = \epsilon$-closure($s_0$)
add $Start$ unmarked to $Dstates$

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

for each input symbol $a$ do
$U = \epsilon$-closure(move($T, a$))
if $U$ is not in $Dstates$ then
add $U$ to $Dstates$ unmarked
$Dtran[T, a] = U$
endfor
endwhile

Each state in $D$ corresponds to a set of states in $N$.
Up to $2^{|\mathcal{V}|}$ 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.

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>

Example subset construction

$NFA$ for $a^*b$

$$
\begin{align*}
&1 
\xrightarrow{e} 2 
\xrightarrow{a} 3 
\xrightarrow{e} 4 
\xrightarrow{e} 5 
\xrightarrow{b} 6 
\end{align*}
$$

$\epsilon$-closure($\cdot$) = 

<table>
<thead>
<tr>
<th>$move(1, e)$</th>
<th>$\epsilon$-closure($\cdot$) =</th>
</tr>
</thead>
<tbody>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$move(2, a)$</td>
<td>$\epsilon$-closure($\cdot$) =</td>
</tr>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$move(3, e)$</td>
<td>$\epsilon$-closure($\cdot$) =</td>
</tr>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$move(4, e)$</td>
<td>$\epsilon$-closure($\cdot$) =</td>
</tr>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$move(5, b)$</td>
<td>$\epsilon$-closure($\cdot$) =</td>
</tr>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>$move(6, e)$</td>
<td>$\epsilon$-closure($\cdot$) =</td>
</tr>
<tr>
<td>${}$</td>
<td>${}$</td>
</tr>
</tbody>
</table>
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

```
S -----> a -----> T -----> b -----> R
```

Initial partition

- accepting = {} =
- non-accepting = {} =

Split groups

\[
\begin{align*}
\text{(state, input) = group} \\
(S, a) = & \\
(T, a) = & \\
(S, b) = & \\
(T, b) = & \\
\end{align*}
\]

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
  
if then then then = else;
  
else else = then;

significant blanks

- FORTRAN and Algo68 ignore blanks
  
do 10 i = 1,25
  
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

Summary

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

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),D09E1
5     100      FORMAT(4H)=(3)
6     200      FORMAT(4 )=(3)
7      D09E1=1
8      D09E1=1,2
9       IF(X)=1
10      IF(X)M=1
11     IF(X)300,200
12    300      CONTINUE
13      END
C     this is a comment
   $ FILE(1)
14      END
```