Scanner

Front end

Source code

Scanner

Parser

IR

Back End

Machine code

Errors
We want to avoid writing scanners by hand
- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA

Scanner Generator

Specifications written as "regular expressions"

Represent words as indices into a global table

source code → Scanner → tokens

Scanner Generator

tables or code
Regular Expressions

- **Regular Expression (over alphabet $\Sigma$)**
  - $\epsilon$ is a RE denoting the set $\{\epsilon\}$
  - If $a$ is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
  - If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
    - $x | y$ is an RE denoting $L(x) \cup L(y)$
    - $xy$ is an RE denoting $L(x)L(y)$
    - $x^*$ is an RE denoting $L(x)^*$
Regular Expression – Example

- **RE for recognizing register names**
  
  \[ \text{Register} \rightarrow r \ (0|1|2| \ldots \mid 9) \ (0|1|2| \ldots \mid 9)^* \]
  
  - Allows registers of arbitrary number
  - Requires at least one digit

- **RE corresponds to a recognizer (or DFA)**

\[ S_0 \overset{r}{\rightarrow} S_1 \overset{(0|1|2| \ldots |9)}{\rightarrow} S_2 \]

Recognizer for \text{Register}

*Transitions on other inputs go to an error state, \( s_e \)
Non-deterministic Finite Automata (NFA)

- Each RE corresponds to a *deterministic finite automaton* (DFA)
  - May be hard to directly construct the right DFA
  - NFA for RE such as \((a \mid b)^* \text{abb}\)

- NFA is a little different from DFA
  - \(S_0\) has a transition on \(\varepsilon\)
  - \(S_1\) has two transitions on \(a\)
Token Recognizer

- Tokens are recognized by NFA

```
main recognizer

1 -> digit -> 2
2 -> digit, other -> 3
3 -> digit, other -> other
4 -> num (real)
5 -> num (integer)
6 -> digit, other -> 7
7 -> digit, other -> other
8 -> num (integer)
9 -> letter, other -> 10
10 -> letter, other -> 1
11 -> id_or_keyword

keyword recognizer

9 -> = -> 12
12 -> other, other -> 13
13 -> equal
14 -> re-read the string stored in the buffer
15 -> program
16 -> integer
17 -> id
```
Automating Scanner Construction

- **RE → NFA** *(Thompson’s construction)*
  - Build an NFA for each term
  - Combine them with \( \varepsilon \)-moves

- **NFA → DFA** *(subset construction)*
  - Build the simulation

- **DFA → Minimal DFA**
  - Hopcroft’s algorithm

- **DFA → RE** *(Not part of the scanner construction)*
  - All pairs, all paths problem
  - Take the union of all paths from \( s_0 \) to an accepting state
**Key idea**

- NFA pattern for each symbol & each operator
- Join them with $\varepsilon$ moves in precedence order

---

**NFA for $a$**

$S_0 \xrightarrow{a} S_1$

**NFA for $ab$**

$S_0 \xrightarrow{a} S_1 \xrightarrow{\varepsilon} S_3 \xrightarrow{b} S_4$

**NFA for $a \mid b$**

$S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_3 \xrightarrow{\varepsilon} S_4$

**NFA for $a^*$**

$S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_3 \xrightarrow{\varepsilon} S_4$

---

Ken Thompson, CACM, 1968
Example of Thompson's Construction

Let's try $a (b \mid c)^*$

1. $a$, $b$, & $c$

2. $b \mid c$

3. $(b \mid c)^*$
Example of Thompson’s Construction (con’t)

4. \( a ( b \mid c )^* \)

Of course, a human would design something simpler ...

But, we can automate production of the more complex one ...
NFA $\rightarrow$ DFA with Subset Construction

- Need to build a simulation of the NFA
- Two key functions
  - $\text{Move}(s_i, a)$ is set of states reachable from $s_i$ by $a$
  - $\varepsilon$-closure($s_i$) is set of states reachable from $s_i$ by $\varepsilon$
- The algorithm:
  - Start state derived from $s_0$ of the NFA
  - Take its $\varepsilon$-closure $S_0 = \varepsilon$-closure($s_0$)
  - Take the image of $S_0$, $\text{Move}(S_0, \alpha)$ for each $\alpha \in \Sigma$, and take its $\varepsilon$-closure
  - Iterate until no more states are added

Sounds more complex than it is...
**Conversion NFA to DFA**

What about \( a \ (b \ | \ c)^* \)?

First, the subset construction: NFA \( \rightarrow \) DFA

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( q_0 )</td>
<td>( q_1, q_2, q_3, q_4, q_6, q_9 )</td>
<td>none</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( q_1, q_2, q_3, q_4, q_6, q_9 )</td>
<td>none</td>
<td>( q_5, q_8, q_9, q_3, q_4, q_6 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( q_5, q_8, q_9 )</td>
<td>none</td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( q_7, q_8, q_9 )</td>
<td>none</td>
<td>( s_2 )</td>
</tr>
</tbody>
</table>

Final states
DFA Minimization

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_0)</td>
<td>({s_1, s_2, s_3} {s_0})</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

To produce the minimal DFA
Another Example

Remember \((a \mid b)^* \text{abb}\) ?

Applying the subset construction:

<table>
<thead>
<tr>
<th>Iter.</th>
<th>State</th>
<th>Contains</th>
<th>(\varepsilon)-closure(move((s_i, a)))</th>
<th>(\varepsilon)-closure(move((s_i, b)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(s_0)</td>
<td>(q_0, q_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
<tr>
<td>1</td>
<td>(s_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_3)</td>
</tr>
<tr>
<td></td>
<td>(s_2)</td>
<td>(q_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
<tr>
<td>2</td>
<td>(s_3)</td>
<td>(q_1, q_3)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_4)</td>
</tr>
<tr>
<td>3</td>
<td>(s_4)</td>
<td>(q_1, q_4)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
</tbody>
</table>

Iteration 3 adds nothing to \(S\), so the algorithm halts.

Our first NFA contains \(q_4\) (final state)
Another Example (cont’d)

The DFA for \((a \mid b)^* \text{abb}\)

- Not much bigger than the original
- All transitions are deterministic

\[\begin{array}{c|cc}
\delta & a & b \\
\hline
s_0 & s_1 & s_2 \\
\hline
s_1 & s_1 & s_3 \\
\hline
s_2 & s_1 & s_2 \\
\hline
s_3 & s_1 & s_4 \\
\hline
s_4 & s_1 & s_2 \\
\end{array}\]
Another Example (cont’d)

Applying the minimization algorithm to the DFA

<table>
<thead>
<tr>
<th></th>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>${s_4} {s_0,s_1,s_2,s_3}$</td>
<td>${s_4}$</td>
<td>$s_4$</td>
<td>none</td>
<td>${s_0, s_1, s_2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${s_0,s_1,s_2,s_3}$</td>
<td></td>
<td></td>
<td>${s_3}$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>${s_4} {s_3} {s_0,s_1,s_2}$</td>
<td>${s_0,s_1,s_2}$</td>
<td>$s_3$</td>
<td>none</td>
<td>${s_0, s_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${s_3}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>${s_4} {s_3} {s_1} {s_0,s_2}$</td>
<td>${s_0,s_2}</td>
<td>s_1}$</td>
<td>$s_1$</td>
<td>none</td>
</tr>
</tbody>
</table>

**final state**
Building Faster Scanners from the DFA

- **Table-driven recognizers waste effort**
  - Read (& classify) the next character
  - Find the next state
  - Assign to the state variable
  - Trip through case logic in $\delta()$ & $\text{action()}$
  - Branch back to the top

- **We can do better**
  - Encode state & actions in the code
  - Do transition tests locally
  - Generate ugly, spaghetti-like code
  - Takes (many) fewer operations per input character

```cpp
char \leftarrow \text{next character};
state \leftarrow s_0;
call action(state,char);
while (char \neq \text{eof})
  state \leftarrow \delta(state,char);
call action(state,char);
char \leftarrow \text{next character};
if T(state) = \text{final} then
  report acceptance;
else
  report failure;
```
Building Faster Scanners from the DFA

**A direct-coded recognizer for \( r \ Digit Digit^* \)**

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

```
goto s_0;
s_0: word \leftarrow \emptyset;
    char \leftarrow \text{next character};
    if (char = 'r')
        then goto s_1;
        else goto s_e;
s_1: word \leftarrow word + char;
    char \leftarrow \text{next character};
    if ('0' \leq char \leq '9')
        then goto s_2;
        else goto s_e;
s_2: word \leftarrow word + char;
    char \leftarrow \text{next character};
    if ('0' \leq char \leq '9')
        then goto s_2;
        else if (char = \text{eof})
            then report success;
            else goto s_e;
s_e: print error message;
    return failure;
```
Summary

- **Building scanner**
  - All this technology automates scanner construction
  - Implementer writes down the regular expressions
  - Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
  - This reliably produces fast, robust scanners

- **For most modern language features, this works**
  - You should think twice before introducing a feature that defeats a DFA-based scanner
    - insignificant blanks (Fortran: \texttt{anint = an int = an int})
    - non-reserved keywords (e.g. \texttt{int if = 1;})