Lexical Analysis

- elim. white space, comments
- group chars into tokens
- speed is important

Lexeme:
string of chars matched
for a token

Token:
data structure containing
token type and value
Types of tokens

Values
1, 3.14, true, 'c', "abc",

Identifiers
x, y2, x42,

Keywords
if, while,

Symbols
+,-, <, <=, ;
Example

Input:

\[ x = y \times 5; \]

Output:

\[ \text{ID}(x), \text{ASSIGN}, \text{ID}(y), \text{MUL}, \text{INTCONST}(5), \text{SEM} \]
Idea

- describe lexemes as regular expressions (REs)
- translate REs into NFA
- translate NFA into DFA
- implement table-driven DFA
- use JLex to translate REs into DFA
Regular Expressions

symbols
alternation
concatenation
repetition
parentheses
nothing

Example:

Identifier:

(a|z|A|z|2)
(a|z|A|z|2|0|1|9|_|$)*
Abbreviations

- ab/c
- [abcd]
- [a-z]
- [~X]
- x?
- x+
- u . u
- 1 +

- (a b)c
- (a b c d)
- (a l ... l z)
- anything but x
- x | 3
- x (x++)
- the string itself
- u + u
- anything but \n
Compiler

Error Msg / Error Msg . java
Parse / Lexer . java
Parse / Main . java
Parse / sym . java
Parse / Tiger . lex
package Parse;

import ErrorMsg.ErrorMsg;

%%
%
%
%
%
%
%

% function nextToken

digits = [0-9]+
%
%
if { ... }

...

{ ... }
What's a DFA?
DFA implementation

2-dim table

| state | a | b | c | d | e |...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DFA implementation

state = 0;
while (!end-of-file()) {
    switch (state) {
        case 0: break;
        case 1: state = 17; break;
        ...
    }
}

3
3
DFA implementation

state0:

    goto state1;

state1:

    goto state1;

end:
What's an NFA?
RE $\rightarrow$ NFA (Thompson's construction)
RE → NFA (Appel's construction)

- $E \xrightarrow{\varepsilon} O$
- $a \xrightarrow{\varepsilon} O$

$N \cup M$

$N \mid M$

$M^*$
Example

$$(a|b)^*abb$$
NFA $\rightarrow$ DFA

Set of NFA states = DFA state
NFA → DFA Translation

RE: \((a|b)^*abb\)

NFA:

DFA:

\[
\begin{align*}
A &= \{1, 2, 3, 5, 8\} \\
B &= \{4, 9, 2, 3, 5, 7, 8\} \\
C &= \{6, 2, 3, 5, 7, 8\} \\
D &= \{6, 10, 2, 3, 5, 7, 8\} \\
E &= \{6, 11, 2, 3, 5, 7, 8\}
\end{align*}
\]

\[
\begin{array}{c|cc}
   & a & b \\
\hline
A & A & B \\
B & B & C \\
C & B & D \\
D & B & E \\
E & B & C \\
\end{array}
\]
**NFA→DFA Algorithm**

*E-closure*

In: set of states $S$
Out: set of states that can be reached with $E$-edges from $S$

---

**DFA-Edge**

In: set of states $S$
input symbol $c$
Out: set of states $T$, s.t. $S \xrightarrow{c} T$

From states in $S$:
- follow all transitions on $c$
- then calculate $E$-closure
NFA$\rightarrow$DFA Algorithm (cont.)

Start-state of DFA = $\varepsilon$-closure(start state of NFA);

loop
pick DFA state $S$ and input $c$;
$T = \text{DFA-Edge}(S, c)$;
if ($T$ didn't exist yet)
   add state $T$ to DFA;
   add edge $S \xrightarrow{c} T$;
until (no more edge can be added)
JLex Translation

\[
\text{if} \quad [a-zA-Z][a-zA-Z0-9]* \quad \{ \text{return IF;}; \}\ \\
. \quad \{ \text{return ID;}; \} \quad \{ \text{error ()}; \}
\]
JLex Strategy

- rule priority
  rule for ID has to be after keywords

- longest match
  the rule that matches more characters wins
Recognizing the Longest Match

Example from p. 24

if |--not-a-comment

most recent final state

Current position of automaton

Input position at last call to lexical analyzer
Start States

- allow breaking up the recognition of a token into multiple REs
- allow additional computation for complicated input

Example (p. 33):

```%
% state COMMENT
%
\%
<YYINITIAL> if    \{ ... \}
<YYINITIAL> "(*)" \{ yybegin(COMMENT); 
<COMMENT> "(*"   \{ yybegin(YYINITIAL); 
<COMMENT> .     \{ 3
```
A RE not prefixed
by a \texttt{(<STATE>)}
operates in all states.
**DFA Optimization**

**Given:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B D</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>B C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>B E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>B C</td>
<td></td>
</tr>
</tbody>
</table>

**Find:** optimized table

**E.g.:** A and C look the same
Idea: combine identical rows

works for A and C above

 Doesn't work for

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y Z</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Z X</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>X Y</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
+ ---- a ---- y ---- b ---- a ---- z ---- b ---- a ---- y ---- b ---- a
|     | a   |     | a   | a   | a   | a   | a   | a   | a   |
|     | b   |     | b   |     | b   |     | b   |     | b   |
```

```
2a x
```

```
b
```
DFA Optimization Algorithm

- Combine all final states into one.

- Combine all non-final states into one.

- Split a group of states that violates the grouping.

- Repeat the previous state until no more splits necessary.
### Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABCD</td>
<td>ABCD</td>
</tr>
<tr>
<td>B</td>
<td>ABCD</td>
<td>ABCD</td>
</tr>
<tr>
<td>C</td>
<td>ABCD</td>
<td>ABCD</td>
</tr>
<tr>
<td>D</td>
<td>ABC  E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>ABCD</td>
<td>ABCD</td>
</tr>
</tbody>
</table>

- **Split**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>B</td>
<td>ABCD</td>
<td>ABCD</td>
</tr>
<tr>
<td>C</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>D</td>
<td>ABC  E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>ABC</td>
<td>ABC</td>
</tr>
</tbody>
</table>

- **Split off**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>AC</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>AC</td>
</tr>
</tbody>
</table>

- **Done**
Solution:

\[(a1b)^* \, a \, bb\]

\[
\downarrow
\]

NFA

\[
\downarrow
\]

DFA

---

The diagram illustrates a Non-Deterministic Finite Automaton (NFA) or a Deterministic Finite Automaton (DFA) with states labeled as AC, B, D, and E, connected by transitions labeled with 'a' and 'b'.
Def. i Language

Given: an alphabet (e.g., ASCII)

A language is the set of all valid strings over the alphabet.
Example Languages

(ab)*abb \{abb, aabb, babb, aaabb, ababb, baabb, bbabb, \ldots\}

0[1-9][0-9]* IN

? Java

? English
### Classification of Languages

(by Noam Chomsky, MIT)

<table>
<thead>
<tr>
<th>Language</th>
<th>Tool</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>RE</td>
<td>scanning</td>
</tr>
<tr>
<td>context-free</td>
<td>BNF</td>
<td>parsing</td>
</tr>
<tr>
<td>context-sensitive</td>
<td>rewrite systems</td>
<td>sem. &amp; qual.</td>
</tr>
<tr>
<td>unrestricted</td>
<td>Turing Machine</td>
<td></td>
</tr>
</tbody>
</table>
Limitations of Languages

"Regular lang. can't count."
\[ a^n b^n \] not regular
\[ A..A..*/.*/ \]

"Context-free languages can't remember counts."
\[ a^n b^n c^n \] not context-free

```c
int foo(int, int);
int i = foo(1, 2);
int j = foo(3, 4);
```
Difficult Scanning Problems

- nested comments ($a^n b^n$)
- strings with escape chars,
- PL/I:
  IF IF = THEN THEN THEN ELSE
- FORTRAN:
  DO 20 I = 1, 10
  CONTINUE
- C++:
  
  ```
  C x (int);
  C y (5);
  C z (a);
  ```
Summary

- lex, anal, split from parsing to make parser simpler
- set of valid lexemes is a regular language
- lexemes described by REs
- REs translated to NFA
- NFA translated to DFA
- DFA optimized
- DFA implemented with tables
- RE → NFA → DFA → opt. DFA → tables automated with lex/flex/JLex
- JLex uses rule priority/longest match
- JLex offers start states for scanning non-regular constructs