Lexical Analysis - An Introduction
The Front End

Source code → Front End → IR → Back End → Machine code

The front end is not monolithic
The Front End

Scanner
• Maps stream of characters into words
  → Basic unit of syntax
  → \( x = x + y \); becomes set of tokens \(<\text{type}, \text{lexeme}>\)
    \(<\text{id},x> <\text{eq},=> <\text{id},x> <\text{pl},+> <\text{id},y> <\text{sc},;>\)
Where is Lexical Analysis Used?

For traditional languages but where else...

• Web page “compilation”
  • Lexical Analysis of HTML, XML, etc.
• Natural Language Processing
• Game Scripting Engines
• OS Shell Command Line
• GREP
• Prototyping high-level languages
• JavaScript, Perl, Python
The Big Picture

Why study lexical analysis?
- We want to avoid writing scanners by hand
- We want to harness the theory from classes like CISC 303

Goals:
- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies

Diagram:
- Source code → Scanner
- Specifications → Scanner Generator
- Scanner Generator → Regular Expressions
- Scanner Generator → tables or code
- Scanner → parts of speech & words
Regular Expressions

Powerful notation to specify lexical rules

• Simplifies scanner construction

• Notation describes set of strings over some alphabet

• Entire set of strings called the language

• If r is an RE, L(r) is the language it specifies
Regular Expressions (more formally)

- Over some alphabet $\Sigma$
- $\varepsilon$ is a RE denoting the empty set
- If $a$ is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
Regular Expressions (more formally)

Given sets $R$ and $S$

- **Closure**: $R^*$ is an RE denoting
  \[ \bigcup_{0 \leq i \leq \infty} R^i \]

- **Concatenation**: $RS$ is an RE denoting
  \[ \{st \mid s \in R \text{ and } t \in S\} \]

- **Alternation**: $R | S$ is an RE denoting
  \[ \{s \mid s \in R \text{ or } s \in S\} \]
  - Often written $R \cup S$

Note: Precedence is closure, then concatenation, then alternation
Examples of Regular Expressions

Identifiers:

\[
\text{Letter} \rightarrow (a|b|c| \ldots |z|A|B|C| \ldots |Z) \\
\text{Digit} \rightarrow (0|1|2| \ldots |9) \\
\text{Identifier} \rightarrow \text{Letter} (\text{Letter} | \text{Digit})^*
\]

Numbers:

\[
\text{Integer} \rightarrow (+|-|\varepsilon) (0| (1|2|3| \ldots |9)(\text{Digit}^*)) \\
\text{Decimal} \rightarrow \text{Integer} . \text{Digit}^* \\
\text{Real} \rightarrow (\text{Integer} | \text{Decimal}) \varepsilon (+|-|\varepsilon) \text{Digit}^* \\
\text{Complex} \rightarrow (\text{Real} , \text{Real})
\]

Numbers can get much more complicated!
Regular Expressions (the point)

REs are used to specify the words to be translated to parts of speech by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers (i.e. scanners) from regular expressions.

You may have seen this construction in a Automata Course.

⇒ We study REs and associated theory to automate scanner construction!
What is the regular expression for a register name?

Examples: $r1, r25, r999 \iff$ These are OK.

$r, s1, a25 \iff$ These are not OK.
Consider the problem of recognizing register names

\[ \text{Register} \rightarrow r \ (0|1|2| \ldots \ | 9) \ (0|1|2| \ldots \ | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit
Finite Automaton (FA)

- An abstract machine that corresponds to a particular RE
- Recognizers can scan a stream of symbols to find words

Transition Diagram for Number

- $S_0$: initial state
- $S_1$: transition for 0
- $S_2$: transition for (0|1|2|...|9)

Accepting states:
Finite Automaton (FA)

An FA is a five-tuple \((S, \Sigma, \delta, s_0, S_F)\) where

- \(S\) is the set of states
- \(\Sigma\) is the alphabet
- \(\delta\) is a set of transition functions
  - takes a state and a character and returns a new state
- \(s_0\) is the start state
- \(S_F\) is the set of final states
Finite Automaton (FA)

Transition Diagram for *Number*
Consider the problem of recognizing register names

\[ \text{Register} \rightarrow r \ (0|1|2| \ldots \ | 9) \ (0|1|2| \ldots \ | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

What does the DFA look like?
Consider the problem of recognizing register names

\[ Register \rightarrow r \ (0|1|2| \ldots \ | 9) \ (0|1|2| \ldots \ | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

\[(0|1|2| \ldots \ 9)\]

**Recognizer for Register**

*Transitions on other inputs go to an error state, \( s_e \)*
DFA operation

- Start in state $S_0$ & take transitions on each input character
- DFA accepts a word $x$ iff $x$ leaves it in a final state ($S_2$)

So,

- $r_{17}$ takes it through $s_0, s_1, s_2$ and accepts
- $r$ takes it through $s_0, s_1$ and fails
- $a$ takes it straight to $s_e$
Example (continued)

To be useful, recognizer must turn into code

Char $\leftarrow$ next character
State $\leftarrow s_0$

while (Char $\neq$ EOF)
    State $\leftarrow \delta$(State,Char)
    Char $\leftarrow$ next character

if (State is a final state) then report success
else report failure

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Skeleton recognizer

Table encoding RE
What if we need a tighter specification?

Digit Digit* allows arbitrary numbers
- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31?

Write a tighter regular expression

→ Register → r ( (0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31) )
→ Register → r0|r1|r2| ... |r31|r00|r01|r02| ... |r09

Produces a more complex DFA
- Has more states
- Same cost per transition
- Same basic implementation
Tighter register specification (continued)

The DFA for

\[
\text{Register} \rightarrow r \ ( (0|1|2) (Digit \mid \varepsilon) \mid (4|5|6|7|8|9) \mid (3|30|31) )
\]

- Accepts a more constrained set of registers
- Same set of actions, more states
## Tighter register specification (continued)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

**Table encoding RE for the tighter register specification**

Runs in the same skeleton recognizer
Extra Slides
The Front End

Parser

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

*We’ll come back to parsing in a couple of lectures*
## Set Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union of L and M</strong></td>
<td>( L \cup M = {s \mid s \in L \text{ or } s \in M } )</td>
</tr>
<tr>
<td>**Written L</td>
<td>M**</td>
</tr>
<tr>
<td><strong>Concatenation of L and M</strong></td>
<td>( LM = {st \mid s \in L \text{ and } t \in M } )</td>
</tr>
<tr>
<td><strong>Written LM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of L</strong></td>
<td>( L^* = \bigcup_{0 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Positive Closure of L</strong></td>
<td>( L^+ = \bigcup_{1 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L</strong></td>
<td></td>
</tr>
</tbody>
</table>

*These definitions should be well known*