Lexical Analysis - An Introduction
The purpose of the front end is to deal with the input language

- Perform a membership test: \( \text{code} \in \text{source language?} \)
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

*The front end is not monolithic*
The Front End

Scanner

- Maps stream of characters into words
  - Basic unit of syntax
  - $x = x + y$ becomes $\langle id, x \rangle \langle eq, = \rangle \langle id, x \rangle \langle pl, + \rangle \langle id, y \rangle \langle sc, ; \rangle$

- Characters that form a word are its lexeme
- Its part of speech (or syntactic category) is called its token type
- Scanner discards white space & (often) comments

Source code $\rightarrow$ Scanner $\rightarrow$ tokens $\rightarrow$ Parser $\rightarrow$ IR $\rightarrow$ Errors

Speed is an issue in scanning
⇒ use a specialized recognizer
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
The Big Picture

• Language syntax is specified with *parts of speech*, not *words*.  
• Syntax checking matches *parts of speech* against a grammar.

1. goal \( \rightarrow \) expr
2. expr \( \rightarrow \) expr op term
3. | term
4. term \( \rightarrow \) number
5. | id
6. op \( \rightarrow \) +
7. | –

\[
S = \text{goal} \\
T = \{ \text{number, id, +, -} \} \\
N = \{ \text{goal, expr, term, op} \} \\
P = \{ 1, 2, 3, 4, 5, 6, 7 \}
\]
The Big Picture

- Language syntax is specified with *parts of speech*, not *words*.
- Syntax checking matches *parts of speech* against a grammar.

1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | −

\[ S = \text{goal} \]
\[ T = \{ \text{number, id, +, -} \} \]
\[ N = \{ \text{goal, expr, term, op} \} \]
\[ P = \{ 1, 2, 3, 4, 5, 6, 7 \} \]

No words here!

Parts of speech, not words!
The Big Picture

Why study lexical analysis?

• We want to avoid writing scanners by hand
• We want to harness the theory from other classes

Goals:

→ To simplify specification & implementation of scanners
→ To understand the underlying techniques and technologies

Represent words as indices into a global table

Specifications written as “regular expressions”
Regular Expressions

Lexical patterns form a *regular language*

*** any finite language is regular ***

*Regular expressions* (REs) describe regular languages

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 $L(x)^*$

Precedence is closure, then concatenation, then alternation

Ever type “rm *.o a.out”?
### Set Operations (review)

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

Identifiers:

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

Numbers:

\[ \text{Integer} \rightarrow (\pm|\varepsilon) (0| (1|2|3| \ldots |9)(\text{Digit}^*) ) \]
\[ \text{Decimal} \rightarrow \text{Integer} \times \text{Digit}^* \]
\[ \text{Real} \rightarrow ( \text{Integer} \mid \text{Decimal} ) \times (\pm|\varepsilon) \text{Digit}^* \]
\[ \text{Complex} \rightarrow ( \text{Real} \times \text{Real} ) \]

Numbers can get much more complicated!
Regular expressions can be 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 from regular expressions.

Some of you may have seen this construction for string pattern matching.

⇒ We study REs and associated theory to automate scanner construction!
Consider the problem of recognizing ILOC 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

RE corresponds to a recognizer (or DFA)

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

To be useful, recognizer must turn into code

Char ← next character
State ← s_0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

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

Skeleton recognizer

Table encoding RE

<table>
<thead>
<tr>
<th>δ</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>
To be useful, recognizer must turn into code

Char ← next character
State ← $s_0$

while (Char $\neq$ EOF)
    State ← $\delta$(State,Char)
    perform specified action
    Char ← next character

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

**Skeleton recognizer**

<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>$s_e$ error</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

**Table encoding RE**
What if we need a tighter specification?

$Digit\ Digit^*$ allows arbitrary numbers

- Accepts \texttt{r00000}
- Accepts \texttt{r99999}
- What if we want to limit it to \texttt{r0} through \texttt{r31}? 

Write a tighter regular expression

- $\rightarrow Register \rightarrow r\ (0|1|2)\ (Digit\ |\ \varepsilon)\ |\ (4|5|6|7|8|9)\ |\ (3|30|31)\ )$

- $\rightarrow Register \rightarrow r0|r1|r2| ... |r31|r00|r01|r02| ... |r09$

Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation
The DFA for

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

- 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
Goal

• We will show how to construct a finite state automaton to recognize any RE

• Overview:
  → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    ▪ Requires $\varepsilon$-transitions to combine regular subexpressions
  → Construct a deterministic finite automaton (DFA) to simulate the NFA
    ▪ Use a set-of-states construction
  → Minimize the number of states
    ▪ Hopcroft state minimization algorithm
  → Generate the scanner code
    ▪ Additional specifications needed for details
Non-deterministic Finite Automata

Each RE corresponds to a deterministic finite automaton (DFA) 
• May be hard to directly construct the right DFA

What about an RE such as \((a \mid b)^* abb\) ?

This is a little different
• \(S_0\) has a transition on \(\varepsilon\)
• \(S_1\) has two transitions on \(a\)

This is a non-deterministic finite automaton (NFA)
Non-deterministic Finite Automata

• An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$
• Transitions on $\varepsilon$ consume no input
• To “run” the NFA, start in $s_0$ and **guess** the right transition at each step
  → Always guess correctly
  → If some sequence of correct guesses accepts $x$ then accept

Why study NFAs?
• They are the key to automating the RE $\rightarrow$ DFA construction
• **We can paste together NFAs with $\varepsilon$-transitions**
Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no $\varepsilon$ transitions
- DFA’s transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

$\rightarrow$ Obviously

NFA can be simulated with a DFA (less obvious)

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream
Automating Scanner Construction

To convert a specification into code:
1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators
- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser (define all parts of speech)
- You could build one in a weekend!
Automating Scanner Construction

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

NFA $\rightarrow$ DFA (*subset construction*)
- Build the simulation

DFA $\rightarrow$ Minimal DFA
- Hopcroft’s algorithm

DFA $\rightarrow$ RE (*Not part of the scanner construction*)
- All pairs, all paths problem