Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Second homework (bottom-up allocation & instruction scheduling) is due on Friday, February 15, 11:59pm

• First project has been posted. Due dates: CODE Monday March 4; REPORT Wednesday March 6

• Third homework will be posted by Monday

• Midterm: Wednesday, March 13, in class; closed book, closed notes, 80 minutes
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/tokens
  - Basic unit of syntax
  - $x = x + y$; becomes `<id,x><eq,=><id,x><pl,+><id,y><sc,;>`

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

Source code $\xrightarrow{\text{Scanner}}$ tokens $\xrightarrow{\text{Parser}}$ IR

Errors

Speed is an issue in scanning
⇒ use a specialized recognizer
The Front End

**Parser**

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

We’ll get to parsing in the next lectures
The Big Picture

- Language syntax is specified over parts of speech (tokens)
- Syntax checking matches sequence of tokens against a grammar
- Here is an example context free grammar (CFG) $G$:

1. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. $\mid term$
4. $term \rightarrow number$
5. $\mid id$
6. $op \rightarrow +$
7. $\mid -$  

$S = goal$

$T = \{ \text{number, id, +, -} \}$

$N = \{ goal, expr, term, op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$G$ in BNF form $G = (S, T, N, P)$
Why study lexical analysis?

- We want to avoid writing scanners by hand

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

Specifications written as “regular expressions”

Represent words as indices into a global table
Lexical patterns form a regular language

*** any finite language is regular ***

Regular expressions (REs) describe regular languages

Regular Expression (over an alphabet $\Sigma$, a finite set of symbols):

- $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
- 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
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - $xy$ is an RE denoting $L(x)L(y)$
  - $x^*$ is an RE denoting $L(x)^*$
  - $(x)$ is an RE denoting $L(x)$

Precedence is closure, then concatenation, then alternation

Ever type “rm *.o a.out” ?
## 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 $L \cup M$</td>
<td></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>Written $LM$</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>Written $L^*$</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>Written $L^+$</td>
<td></td>
</tr>
</tbody>
</table>

These definitions should be well known.
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:

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

Numbers can get much more complicated!
Regular expressions can be used to specify the words to be translated to parts of speech (tokens) by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions.

⇒ 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,

- $r17$ 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 error state $s_e$ (not shown here)
To be useful, recognizer must turn into code

\[
\begin{align*}
\text{Char} &\leftarrow \text{next character} \\
\text{State} &\leftarrow s_0 \\
\text{while} \ (\text{Char} \neq \text{EOF}) &\quad \text{State} \leftarrow \delta(\text{State}, \text{Char}) \\
&\quad \text{Char} \leftarrow \text{next character} \\
\text{if} \ (\text{State} \text{ is a final state}) &\quad \text{then report success} \\
&\quad \text{else report failure}
\end{align*}
\]

\[\delta\]

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

```
Char ← next character
State ← s₀

while (Char ≠ EOF)
    State ← δ(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>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>s₀</td>
<td>s₁ start</td>
<td>sₑ error</td>
<td>sₑ error</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ error</td>
<td>s₂ add</td>
<td>sₑ error</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ error</td>
<td>s₂ add</td>
<td>sₑ error</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ error</td>
<td>sₑ error</td>
<td>sₑ error</td>
</tr>
</tbody>
</table>

**Table encoding RE**
r 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

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

\[ \text{Register} \rightarrow r0|r1|r2| ... |r31|r00|r01|r02| ... |r09 \]

Produces a more complex DFA

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

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

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

<table>
<thead>
<tr>
<th>δ</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>s₀</td>
<td>s₁</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₐ</td>
<td>s₂</td>
<td>s₂</td>
<td>s₅</td>
<td>s₄</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₐ</td>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₃</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₄</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₅</td>
<td>sₐ</td>
<td>s₆</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₆</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
<tr>
<td>s₇</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
<td>sₐ</td>
</tr>
</tbody>
</table>

*Table encoding RE for the tighter register specification*

Runs in the same skeleton recognizer
The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
More Lexical Analysis

Syntax Analysis (top-down)

Read EaC: Chapter 3.1 – 3.3