CS415 Compilers

Lexical Analysis

Part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
• Second homework due this Friday, February 18.

• First project (local instruction scheduler) has been posted
  Deadline for code: March 2
  Deadline for report: March 4

• First midterm: Wednesday, February 23
  In class exam, 60 minutes,
  Topics: ILOC, instruction scheduling, register allocation

• Spring recess: March 12 - 20

• Final exam (exam code C): Tuesday, May 10
  1:00pm - 2:00pm
  In person, location TBD
The purpose of the front end is to deal with the input language
• Perform a membership test: code ∈ 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*
Scanner

- Maps stream of characters into words/tokens
  - Basic unit of syntax
  - $x = x + y$; becomes $<\text{id},x><\text{eq},>=<\text{id},x><\text{pl},+><\text{id},y><\text{sc},;>$

- Character sequence that forms a word/token is its lexeme

- Its part of speech (or syntactic category) is called its token type

- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer
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*
• 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. $term \ | \ term$
4. $term \rightarrow number$
5. $term \ | \ id$
6. $op \rightarrow \ +$
7. $op \ | \ -$

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

$G$ 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
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”
The Big Picture

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

Represent words as indices into a global table

Specifications written as "regular expressions"

source code → Scanner → parts of speech & words (tokens)

specifications → Scanner Generator → tables or code
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
  - \( x | y \) 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
### Set Operations

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

*These definitions should be well known*
Examples of Regular Expressions

Identifiers:

\[\text{Letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)\]

\[\text{Digit} \rightarrow (0 \mid 1 \mid 2 \mid \ldots \mid 9)\]

\[\text{Identifier} \rightarrow \text{Letter} ( \text{Letter} \mid \text{Digit} )^*\]

Numbers:

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

\[\text{Decimal} \rightarrow \text{Integer} \cdot \text{Digit}^*\]

\[\text{Real} \rightarrow (\text{Integer} \mid \text{Decimal}) \varepsilon (\pm \mid \varepsilon) \text{Digit}^*\]

\[\text{Complex} \rightarrow (\text{Real} \mid \text{Real})\]

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)

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

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

\[ \delta(s_x, a) = s_y \]

<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**
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)  
perform specified action  
Char $\leftarrow$ 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>$s_e$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>start</td>
<td>error</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>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
→ Register → r ( (0|1|2) (Digit | \(\varepsilon\)) | (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
The DFA for
\[ \text{Register} \rightarrow r ( (0|1|2) \text{ (Digit } \cup \varepsilon) \text{ (4|5|6|7|8|9) (3|30|31) } ) \]

- Accepts a more constrained set of registers
- Same set of actions, more states
<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
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