Lexical Analysis

Lecture 3

January 10, 2018
Announcements

- PA1c due tonight at 11:50pm!
  - Don’t forget about PA1, the Cool implementation!
  - Use Monday’s lecture, the video guides and Cool examples if you’re stuck with Cool!
Programming Assignments Going Forward

- C was allowed for PA1, but not for PA2 through PA6
  - How comfortable are we with other languages?
  - Python, Haskell, Ruby, OCaml, and JavaScript
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```java
class Main { ... }
```

Diagram:
```
Lexical Analyzer

CLASS, IDENT, LBRACE, ...
```
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```
class Main {
  ...
```

Lexical Analyzer

```
CLASS, IDENT, LBRACE, ...
```
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```
class Main {
  ...
```

Lexical Analyzer

CLASS, IDENT, LBRACE, ...
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```
class Main { ... }
```

Lexical Analyzer

CLASS, IDENT, LBRACE, ...
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```
class Main { ... }
```

Lexical Analyzer

```
CLASS, IDENT, LBRACE, ...
```
Lexical Analysis Summary

- Lexical analysis turns a stream of characters into a stream of tokens
- Regular expressions are a way to specify sets of strings, which we use to describe tokens

```
class Main {
...
```

Lexical Analyzer

CLASS, IDENT, LBRACE, ...

Compiler Construction
Cunning Plan

- Informal Sketch of Lexical Analysis
  - LA identifies tokens from input string
  - `List<Token> lexer ( char[] )`

- Issues in Lexical Analysis
  - Lookahead
  - Ambiguity

- Specifying Lexers
  - Regular Expressions
  - Examples
Definitions

- **Token** — set of strings defining an atomic element with a distinct meaning
  - a syntactic category

- **Lexeme** — a sequence of characters than can be categorized as a Token
Definitions

- **Token** — set of strings defining an atomic element with a distinct meaning
  - a syntactic category
  - In English:
    - noun, verb, adjective
  - In Programming:
    - identifier, integer, keyword, whitespace, ...

- **Lexeme** — a sequence of characters than can be categorized as a Token
# Tokens and Lexemes

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>class</td>
</tr>
<tr>
<td>LT</td>
<td>&lt;</td>
</tr>
<tr>
<td>FALSE</td>
<td>false</td>
</tr>
<tr>
<td>IDENT</td>
<td>variable_name</td>
</tr>
</tbody>
</table>
## Tokens and Lexemes

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>class</td>
</tr>
<tr>
<td>LT</td>
<td>&lt;</td>
</tr>
<tr>
<td>FALSE</td>
<td>false</td>
</tr>
<tr>
<td>IDENT</td>
<td>variable_name</td>
</tr>
</tbody>
</table>
## Tokens and Lexemes

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>class</td>
</tr>
<tr>
<td>LT</td>
<td>&lt;</td>
</tr>
<tr>
<td>FALSE</td>
<td>false</td>
</tr>
<tr>
<td>IDENT</td>
<td>variable_name</td>
</tr>
</tbody>
</table>

By the way, what do you think of Cool’s fi, pool, esac...?
Context for Lexers

- Lexing and Parsing go hand-in-hand
  - Parser uses distinctions between tokens
    - e.g., a keyword is treated differently than an identifier

```
Lexer
get_char() -> character

Parser
get_token() ->
```

```
input
```
Lexical Analysis

- Consider this example:
  
  ```
  if(i=j) then
    z<-0
  else
    z<-1
  ```

- The input is simply a sequence of characters:
  
  ```
  if(i=j) then\n  tz<-0\n  else ...
  ```

- **Goal** partition input strings into substrings
  
  - Then, classify them according to their role (tokenize!)
Tokens

- Tokens correspond to sets of strings

- **Identifier**— strings of letters or digits, starting with a letter

- **Integer**— a non-empty string of digits

- **Keyword**— “else” or “class” or “let” ...

- **Whitespace**— Non-empty sequence of blanks, newlines, and/or tabs

- **OpenParen**— a left parenthesis (

- **CloseParen**— a right parenthesis )
Building a Lexical Analyzer

- Lexer implementation must do three things
  1. Recognize substrings corresponding to tokens
  2. Return the value of lexeme of the token
  3. Report errors intelligently (line numbers for Cool)
Lexical Analyzer Implementation

- Lexer usually discards “uninteresting” tokens that don’t contribute to parsing

- Examples: Whitespace, comments
  - Exceptions: Which languages care about whitespace?

- Review: What would happen if we removed all whitespace and comments before lexing?
Example

► Recall:

```plaintext
if (i = j) then
  z<-0
else
  z<-1
```

► Our Cool Lexer would return token-lexeme-linenumber tuples

```plaintext
<IF, "if", 1>
<WHITESPACE, " ", 1>
<OPENPAREN, "(", 1>
<IDENTIFIER, "i", 1>
<WHITESPACE, " ", 1>
<EQUALS, "=", 1>
```
Lexing Considerations

- The goal is to partition the input string into meaningful tokens.
  - Scan left to right (i.e., in order)
  - Recognize tokens
Lexing Considerations

- The goal is to partition the input string into meaningful tokens.
  - Scan left to right (i.e., in order)
  - Recognize tokens

- We really need a way to describe the lexemes associated with each token
Lexing Considerations

- The goal is to partition the input string into meaningful tokens.
  - Scan left to right (i.e., in order)
  - Recognize tokens

- We really need a way to describe the lexemes associated with each token
- And also a way to handle ambiguities
  - is “if” two variables “i”, “f”
Regular Languages

Sounds like we can use DFAs to recognize lexemes
  With accepting states corresponding to tokens!

Example: Capture the word “class”

Example: Capture some variable name

A = letter
AN = alphanumerics
W = whitespace
Capturing Multiple Tokens

What about both “class” and variable names?
Lexical Analyzer Generators

- We like regular languages as a means to categorize lexemes into tokens
- We don’t like the complexity of implementing a DFA manually

- We use **Regular Expressions** to describe regular languages
  - And our tokens are recognizable as regular languages!

- **Regular Expressions** can be automatically turned into a DFA for rapid lexing!
Definition Let $\Sigma$ be a set of characters. A language over $\Sigma$ is a set of strings of characters drawn from $\Sigma$. $\Sigma$ is called the alphabet.
Examples of Languages

- Alphabet = English Characters
- Language = English Sentences
  - Note: Not every string on English characters is an English sentence
  - Example: adsfasdklg gdsajkl

- Alphabet = ASCII characters
- Language = C Programs
  - Note: ASCII character set is different from English character set
Notation

- Languages are sets of strings

- We need some notation for specifying which sets we want
  - i.e., which strings are in a set?

- For lexical analysis, we care about regular languages, which can we described using regular expressions
Regular Expressions

- Each *regular expressions* is a notation for a regular language (a set of “words”)
  - Notation forthcoming!

- If $A$ is a regular expression, we write $L(A)$ to refer to the language denoted by $A$
Base Regular Expressions

- **Single character: ‘c’**
  - $L('c') = \{ 'c' \}$

- **Concatenation: $AB$**
  - $A$ and $B$ are both Regular expressions
  - $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$

- **Example:** $L('i' 'f') = \{ 'if' \}$
Compound Regular Expressions

▶ Union

▶ \[ L(A | B) = \{ \text{s} \mid \text{s} \in L(A) \text{ or } \text{s} \in L(B) \} \]

▶ Examples

▶ \[ L(\text{‘if’} \mid \text{‘then’} \mid \text{‘else’}) = \{ \text{‘if’, ‘then’, ‘else’} \} \]
▶ \[ L(\text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’}) = \text{what?} \]

▶ \[ L( (\text{‘0’} \mid \text{‘1’}) (\text{‘0’} \mid \text{‘1’}) ) = \text{what?} \]
So far, base and compound regular expressions only describe **finite languages**

**Iteration:** $A^*$

- $L(A^*) = \{ "\" \} \cup \{ L(A) \} \cup \{ L(AA) \} \cup \{ L(AAA) \} \cup ...$

**Examples**

- $L('0'^*) = \{ """, "0", "00", "000", ... \}$
- $L('1''0'^*) = \{ "1", "10", "100", "1000", ... \}$

**Empty:** $\varepsilon$

- $L(\varepsilon) = \{ "" \}$
Example: Keyword

- Keywords: ‘else’ or ‘if’ or ‘fi’

  ‘else’ | ‘if’ | ‘fi’

  (Recall that ‘else’ abbreviates concatenation of ‘e’ ‘l’ ‘s’ ‘e’ )
Example: Integer

- Integer: a non-empty string of digits

  \[\text{digit} = \text{‘0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’}\]

  \[\text{number} = \text{digit digit*}\]

- Abbreviation: A+ = AA*
Example: Identifiers

- Identifier: string of letters or digits, start with a letter

\[
\begin{align*}
\text{letter} &= \text{‘A’} \mid \ldots \mid \text{‘Z’} \mid \text{‘a’} \mid \ldots \mid \text{‘Z’} \\
\text{ident} &= \text{letter} \ (\text{letter} \mid \text{digit})^* 
\end{align*}
\]

- Is \((\text{letter}^* \mid \text{digit}^*)\) the same?
Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

( ' ' | \t | \n | \r ) +
Example: Phone Numbers

Regular expressions are everywhere!
Consider: (123)-234-4567

\[ \Sigma = \{ 0, 1, 2, \ldots, 9, (, ), - \} \]
area digit digit digit
exch digit digit digit
phone digit digit digit digit
number ‘(’ area ‘)’ ‘-’ exch ‘-’ phone
Example: Email addresses

Consider kjleach@eecs.umich.edu

$$\Sigma \ {a, b, c, \ldots, z, \ '.', \ '@'}$$

name letter+

address name '@' name ('.' name)*
Regular Expression Summary

- Regular expressions describe many useful languages
- Given a string \( s \) and a regexp \( R \), we can find if \( s \in L(R) \)
  - Is this enough?
Regular Expression Summary

- Regular expressions describe many useful languages
- Given a string $s$ and a regexp $R$, we can find if $s \in L(R)$
  - Is this enough?
  - NO Recall we need the original lexeme!
Regular Expression Summary

- Regular expressions describe many useful languages
- Given a string $s$ and a regexp $R$, we can find if $s \in L(R)$
  - Is this enough?
    - NO Recall we need the original lexeme!
- We must adapt regular expressions to this goal
Next time

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata
  - Nondeterministic Finite Automata
- Implementation of Regular Expressions
  - Regexp $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ lookup table
- Lexical Analyzer Generation (i.e., doing this all automatically)
Lexical Specification (1)

- Start with a set of tokens (protip, PA2 lists them for Cool)
- Write a regular expressions for the lexemes representing each token
  - Number = digit+
  - IF = “if”
  - ELSE = “else”
  - IDENT = letter ( letter | digit ) *
- ...
Lexical Specification (2)

Construct R, matching all lexemes for all tokens

- $R = \text{Number} \mid \text{IF} \mid \text{ELSE} \mid \text{IDENT} \mid \ldots$
- $R = R_1 \mid R_2 \mid R_3 \mid \ldots$

If $s \in L(R)$, then $s$ is a lexeme

- Also $s \in L(R_j)$ for some $j$
- The particular $j$ corresponds to the type of token reported by lexer
Lexical Specification (3)

- For an input $x_1, \ldots, x_n$
  - Each $x_i \in \Sigma$

- For each $1 \leq i \leq n$, check
  - is $x_1 \ldots x_i \in L(R)$?

- If so, it must be that
  $x_1 \ldots x_i \in L(R_j)$ for some $j$

- Remove $x_1 \ldots x_i$ from input and restart
Example Lexing

\[ R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid \text{Plus} \]

Lex "f + 3 + g"

- "f" matches R (more specifically, Identifier)
- " " matches R (more specifically, Whitespace)

What does the lexer output look like for this example?
Ambiguities

- Ambiguities arise in this algorithm
- \( R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid \text{Plus} \)
- Lex “foo+3”
- “f”, “fo”, and “foo” all match R, but not “foo+”
- How much input do we consume?
  - Maximal munch rule: pick the longest possible substring that matches R
Ambiguities (2)

\[ R = \text{Whitespace} \mid \text{‘new’} \mid \text{Integer} \mid \text{Identifier} \mid \text{Plus} \]

- Lex “new foo”

- “new” matches both the ‘new’ rule and the ‘Identifier’ rule. Which one do you pick?

- Generally, pick the rule listed first
  - Arbitrary, but typical
  - Important for PA2!

‘new’ was listed before ‘Identifier‘, so the token given is ‘new’
Summary

- Regular expressions provide a concise notation for string patterns
- We need to adapt them for lexical analysis to
  - Resolve ambiguities
  - Handle errors (report line numbers)
- Next time, Lexical Analysis Generators
A = letter
AN = alphanumeric
W = whitespace