Administration

• HW1 out later today – due next Monday.

Outline

• Administration
• Compilation in a nutshell (or two)
• What is lexical analysis?
• Writing a lexer
• Specifying tokens: regular expressions
• Writing a lexer generator
  – Converting regular expressions to Non-deterministic finite automata (NFAs)
  – NFA to DFA transformation

Compilation in a Nutshell 1

Source code (character stream)

if (b == 0) a = b;

Lexical analysis

Token stream

if ( b == 0 ) a = b ;

Parsing

Abstract syntax tree (AST)

if b == 0 a = b;

Semantic Analysis

Decorated AST

boolean if int 0 int a int b

int b

int 0

int a

int b

lvalue
Compilation in a Nutshell 2

First step: lexical analysis

Tokens

- Identifiers: x y11 else _i00
- Keywords: if else while break
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1. 1e5 0.e-10
- Symbols: + * {} ++ <= <= [ ] =>
- Strings: “x” “He said, \“Are you?\””
- Comments: /** don’t change this **/

Ad-hoc lexer

- Hand-write code to generate tokens
- How to read code to generate tokens?
  
  ```java
  Token readIdentifier() {
      String id = “”;
      while (true) {
          char c = input.read();
          if (identifierChar(c))
              id = id + String(c);
      }
  }
  ```
Look-ahead character

- Scan text one character at a time
- Use look-ahead character (*next*) to determine what kind of token to read and when the current token ends

```java
char next;
...
while (identifierChar(next)) {
    id = id + String(next);
    next = input.read();
}
```

Ad-hoc lexer: top-level loop

```java
class Lexer {
    InputStream s;
    char next;
    Lexer(InputStream s_) { s = s_; next = s_.read(); }
    Token nextToken() {
        if (identifierChar(next))
            return readIdentifier();
        if (numericChar(next))
            return readNumber();
        if (next == '"') return readStringConst();
        ...
    }
}
```

Preloading `next`. Alternative: define input streams that support lookahead automatically

Problems

- Don’t know what kind of token we are going to read from seeing first character
  - if token begins with “i” is it an identifier?
  - if token begins with “2” is it an integer constant?
  - interleaved tokenizer code is hard to write correctly, harder to maintain
- A more principled approach: *lexer generator* that generates efficient tokenizer automatically (e.g., lex, JLex) from a lexical specification.

Issues

- How to describe tokens unambiguously
  - `2.e0    20.e-01    2.0000``
  - `""    "x"    "\"    "\\ \"
- How to break text up into tokens
  - `if (x == 0) a = x<<1;`
  - `if (x == 0) a = x<1;`
- How to tokenize efficiently
  - tokens may have similar prefixes
  - want to look at each character O(1) times
How to Describe Tokens

- Programming language tokens can be described using regular expressions
- Regular expression $R$ describes a set of strings $L(R)$: $L(R)$ is the "language" defined by $R$
  - $L(abc) = \{ abc \}$
  - $L(\text{hello}|\text{goodbye}) = \{ \text{hello}, \text{goodbye} \}$
  - $L([1-9][0-9]^*) = \text{all positive integer constants}$
  - $L(X(Y|Z)) = L(XY|XZ) = L(XY)$
- Idea: define each kind of token using RE

Regular expression notation

- $a$: an ordinary character stands for itself
- $\epsilon$: the empty string
- $R|S$: any string from either $L(R)$ or $L(S)$:
  \[ L(R|S) = L(R) \cup L(S) \]
- $RS$: string from $L(R)$ followed by one from $L(S)$:
  \[ L(RS) = \{ rs | r \in L(R) \land s \in L(S) \} \]
- $R^*$: zero or more strings from $L(R)$, concatenated
  \[ \varepsilon | R | RR | RRR | RRRR \ldots \]
  ("Kleene star")

Examples

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Strings in $L(R)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>$ab$</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>$a</td>
<td>b$</td>
</tr>
<tr>
<td>$(ab)^*$</td>
<td>&quot;ab&quot; &quot;abab&quot; ...</td>
</tr>
<tr>
<td>$(a</td>
<td>b) \ $</td>
</tr>
</tbody>
</table>

Convenient RE Shorthand

- $R^+$: one or more strings from $L(R)$: $= R(R^*)$
- $R?$: an optional $R$: $= (R|\epsilon)$
- $[\text{abc}]$: one of the listed characters: $(a|b|c|e)$
- $[a-z]$: one char from the range: $(a|b|c|d|e|\ldots)$
- $[^\text{ab}]$: anything but one of the listed chars
- $[^\text{a-z}]$: one character not from the range
- $R\{n\}$: $n$ repetitions of $R$ ($RRRR\ldots$)
- $\backslash x0A$: ASCII 10 (newline)
- $\backslash n$: also newline
### More Examples (JFlex)

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Strings in L(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>digit = [0-9]</code></td>
<td>&quot;0&quot; &quot;1&quot; &quot;2&quot; &quot;3&quot; ...</td>
</tr>
<tr>
<td><code>posint = {digit}+</code></td>
<td>&quot;8&quot; &quot;412&quot; ...</td>
</tr>
<tr>
<td><code>int = -? posint</code></td>
<td>&quot;-42&quot; &quot;1024&quot; ...</td>
</tr>
<tr>
<td><code>real = {int} (. posint)?</code></td>
<td>&quot;-1.56&quot; &quot;12&quot; &quot;1.0&quot;</td>
</tr>
<tr>
<td></td>
<td>= ((-) ε(0</td>
</tr>
<tr>
<td></td>
<td>[a-zA-Z_][a-zA-Z0-9_]* ( C ) identifiers</td>
</tr>
</tbody>
</table>

- Lexer generators support abbreviations – cannot be recursive. Forbidden: `foo = a{foo}|ε`

---

### Zero-width assertions

- Not strictly regular expressions...
- Not supported by all lexer generators.
  - `^R` matches R if preceded by newline
  - `R$` matches R if followed by newline
  - `\b` match a word boundary (Perl)
  - `\A` match beginning of input (Perl)
  - `R_1/R_2` matches R_1 if followed by something matching R_2 (lex)

---

### How to break up text

```plaintext
test = 0;
```

1. `else x = 0`
2. `else x = 0`

- REs alone not enough: need rule for choosing
- Most languages: **longest matching token** wins – even if a shorter token is only way to parse tokens.
  - Exception: early FORTRAN (totally whitespace-insensitive)
  - Ties in length resolved by prioritizing tokens
- RE’s + priorities + longest-matching token rule = lexer definition

---

### Lexer Generator Spec

- **Input to lexer generator:**
  - list of regular expressions in priority order
  - associated **action** for each RE (generates appropriate kind of token, other bookkeeping)
- **Output:**
  - program that reads an input stream and breaks it up into tokens according to the REs.
  - (Or reports lexical error -- "Unexpected character")
Example: JFlex

```plaintext
%line
%column
%%
digits = 0|[1-9][0-9]*
letter = [A-Za-z]
identifier = {letter}{{letter}}[0-9_]*)
whitespace = [\ \t\n\r]+%
{whitespace} /* discard */
{digits}  { return new IntegerConstant(Integer.parseInt(yytext())); }
"if"     { return new IfToken(); }
"while"  { return new WhileToken(); }
...
{identifier}  { return new IdentifierToken(yytext()); }
```

Lexer states

- Most lexer generators allow conditioning on lexer state. Helps with long tokens (strings, comments):

```plaintext
"/*"   { yybegin(COMMENT); }
<COMMENT> {
  "*/"   { yybegin(YYINITIAL); }
  .\n  { /* ignore */ }
}
```

Summary

- Lexical analyzer converts a text stream to tokens
- Ad-hoc lexers hard to get right, maintain
- For most languages, legal tokens conveniently, precisely defined using regular expressions
- Lexer generators generate lexer code automatically from token RE’s, precedence
- Next lecture: how lexer generators work

Groups

- If you don’t have a full group lined up, hang around and talk to prospective group members
- Send mail to cs4120-l if you still cannot make a full group (can also post to cornell.class.cs4120)