Announcements
- Project 1 is 5% of total grade
- Project 2 is 10% of total grade
- Project 3 is 15% of total grade
- Project 4 is 10% of total grade

Today
- Outline of planned topics for course
- Overall structure of a compiler
- Lexical analysis (scanning)
- Syntactic analysis (parsing)

Lexical Analysis (Scanning)

Break character stream into tokens (“words”)
- Tokens, lexemes, and patterns
- Lexical analyzers are usually automatically generated from patterns (regular expressions) (e.g., lex)

Examples

<table>
<thead>
<tr>
<th>token</th>
<th>lexeme(s)</th>
<th>pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>&lt;,&lt;=,=,!,=,...</td>
<td>&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>foo,index</td>
<td>[a-zA-Z_]+[a-zA-Z0-9_]*</td>
<td>[0-9]+</td>
</tr>
<tr>
<td>number</td>
<td>“hi”, “mom”</td>
<td>“.*”</td>
</tr>
<tr>
<td>string</td>
<td>3.14159,570</td>
<td></td>
</tr>
</tbody>
</table>

const pi := 3.14159 => const, identifier(pi), assign, number(3.14159)
Specifying Tokens with SableCC

Theory meets practice:
- Regular expressions, formal languages, grammars, parsing...

SableCC example input file:

```plaintext
Package minijava;
Helpers
digit = ['0'..‘9’];
letter = ['a'..‘z’] | ['A'..‘Z’];
underscore = '_';
not_star = [all - ‘*’];
not_star_slash = [not_star - '/'];
c_comment = '/*' not_star '*(' not_star_slash not_star*)? '*'/';
Tokens
t_plus = '+';
t_if = 'if';
t_id = letter (letter | digit | underscore)*;
t_blank = (‘ ‘ | eol | tab)+;
t_comment = c_comment | line_comment;
Ignored Tokens
t_blank,
t_comment;
```

Recognizing Tokens with DFAs

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>'if'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter (letter</td>
<td>digit)*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter or digit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ambiguity due to matching substrings
- Longest match
- Rule priority

Syntactic Analysis (Parsing)

Impose structure on token stream
- Limited to syntactic structure (⇒ high-level)
- Structure usually represented with an abstract syntax tree (AST)
- Parsers are usually automatically generated from context-free grammars (e.g., yacc, bison, cup, javac, sablecc)

Example

```plaintext
for i = 1 to 10 do
  a[i] = x * 5;

for id(i) equal number(1) to number(10) do
  id(a) lbracket id(i) rbracket equal id(x) times number(5) semi
```

Interaction Between Scanning and Parsing

character stream

Lexical analyzer

Parser

```plaintext
for i = 1 to 10 do
  a[i] = x * 5;

for id(i) equal number(1) to number(10) do
  id(a) lbracket id(i) rbracket equal id(x) times number(5) semi
```
Bottom-Up Parsing: Shift-Reduce

Grammer

\[ a + b + c \]

(1) \( S \rightarrow E \)
(2) \( E \rightarrow E + T \)
(3) \( E \rightarrow T \)
(4) \( T \rightarrow id \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$ a + b + c</td>
<td>shift</td>
</tr>
<tr>
<td>$ a</td>
<td>$ + b + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ + b</td>
<td>$ + c</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>$ +</td>
<td>$ + c</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$</td>
<td>$ + c</td>
<td>shift</td>
</tr>
<tr>
<td>$</td>
<td>$ + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$</td>
<td>$ + T</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ T</td>
<td>$ + T</td>
<td>reduce (1)</td>
</tr>
<tr>
<td>$ S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Rightmost derivation: expand rightmost non-terminals first

SableCC, yacc, and bison generate shift-reduce parsers:
- LALR(1): look-ahead, left-to-right, rightmost derivation in reverse, 1 symbol lookahead
- LALR is a parsing table construction method, smaller tables than canonical LR

Reference: Barbara Ryder's 198:515 lecture notes

Shift-Reduce Parsing Example (precedence problem)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$ a + b + c</td>
<td>shift</td>
</tr>
<tr>
<td>$ a</td>
<td>$ + b + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ + b</td>
<td>$ + c</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>$ +</td>
<td>$ + c</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$</td>
<td>$ + c</td>
<td>shift</td>
</tr>
<tr>
<td>$</td>
<td>$ + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$</td>
<td>$ + T</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ T</td>
<td>$ + T</td>
<td>reduce (1)</td>
</tr>
<tr>
<td>$ S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Grammar with production rules

\[
S: E \{ \$S = \$1; \}
\]
\[
E: E \cdot T \{ \$S = \text{new node("+", \$1, \$3)}; \}
\]
\[
T: T_ID \{ \$S = \text{new leaf("id", \$1)}; \}
\]
\[
\]

Implicit parse tree for \( a + b + c \)

AST for \( a + b + c \)

Reference: Barbara Ryder's 198:515 lecture notes
Using SableCC to specify grammar and generate AST

**Productions**

```
cst_program -> program =
    cst_main_class cst_class_decl* 
    (-> New program(cst_main_class, [cst_class_decl, class_decl]));

cst_exp_list (-> exp*) =
    (empty_rule) cst_exp cst_exp_rest* 
    (-> [cst_exp, cst_expRest]);
   | (empty_rule)
   | (-> []);

cst_exp_rest (-> exp*) =
    t_comma cst_exp (-> [cst_exp, exp]);
```

**Abstract Syntax Tree**

```
program =
    main_class [class_decls]:class_decl*;
exp =
    {call} exp t_id [args]:exp* | ...
```

Parsing Terms

**CFG (Context-free Grammar)**
- production rule
- terminal
- nonterminal
- FOLLOW(X): “the set of terminals that can immediately follow X”

**BNF (Backus-Naur Form) and EBNF (Extended BNF): equivalent to CFGs**

Parsing Terms cont ...

**Top-down parsing**
- LL(1): left-to-right reading of tokens, leftmost derivation, 1 symbol look-ahead
- Predictive parser: an efficient non-backtracking top-down parser that can handle LL(1)
- More generally recursive descent parsing may involve backtracking

**Bottom-up Parsing**
- LR(1): left-to-right reading of tokens, rightmost derivation in reverse, 1 symbol lookahead
- Shift-reduce parsers: for example, bison, yacc, and SableCC generated parsers
- Methods for producing an LR parsing table
  - SLR, simple LR
  - Canonical LR, most powerful
  - LALR(1)

Concepts

**Compilation stages in a compiler**
- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

**Lexical analysis or scanning**
- Tools: SableCC, lex, flex, etc.

**Syntactic analysis or parsing**
- Tools: SableCC, yacc, bison, etc.
Next Time

Lecture
- More undergraduate compilers review

Language Implementation Timeline

For entertainment purposes only!