Undergraduate Compilers Review

Announcements
- Send me email if you have set up a group. Include account names for group.
- Each project is 10% of grade

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

Structure of a Typical Interpreter Compiler

Analysis
- character stream
- tokens "words"
- syntactic analysis
- AST "sentences"
- annotated AST

Synthesis
- IR code generation
- optimization
- code generation
- target language

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>identifier</td>
<td>foo, index</td>
<td>[a-zA-Z_]+[a-zA-Z0-9]*</td>
</tr>
<tr>
<td>number</td>
<td>3.14159, 570</td>
<td>[0-9]*+</td>
</tr>
<tr>
<td>string</td>
<td>&quot;hi&quot;, &quot;mom&quot;</td>
<td>&quot;.*&quot;</td>
</tr>
</tbody>
</table>

const pi := 3.14159 ⇒ const, identifier(pi), assign, number(3.14159)
Interaction Between Scanning and Parsing

Lexical analyzer

character stream

lexer.next() lexor.peek() token

Parser

tree or AST

Specifying Tokens with SableCC

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

SableCC example input file:

```plaintext
Package minijava;

Helpers
all = [0..FFFF];
or = 1;
digit = [0..9];
letter = [a..z] | [A..Z];
underscore = _;
not_star = [all - '*'];
not_star_slash = [not_star - '/'];
c_comment = '/ *' not_star ('*/' | '*')?;
line_comment = '//' not_star_slash;

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

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, javacc, sablecc)

Example

```
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

Lexical analyzer

character stream

lexer

next()

peek()

Lexer

Parser

parse tree or AST

token

Bottom-Up Parsing: Shift-Reduce

Grammer

a + b + c

$ S \rightarrow E$

$ E \rightarrow E + T$

$ E \rightarrow T$

$ T \rightarrow \text{id}$

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

LR Parse Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>id</td>
<td>S</td>
<td>S E T</td>
</tr>
<tr>
<td>0</td>
<td>s4</td>
<td>1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r(3)</td>
<td>r(3)</td>
<td>r(3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r(4)</td>
<td>r(4)</td>
<td>r(4)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r(2)</td>
<td>r(2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Look at current state and input symbol to get action

shift(x): advance input, push n on stack

reduce(k): pop rhs of grammar rule k, k = (lhs :- rhs)

look up state on top of stack and lhs for goto n

push n

accept: stop and success

error: stop and fail

Shift-Reduce Parsing Example

Stack | Input | Action | Stack | Input | Action |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 0</td>
<td>a + b + c</td>
<td>shift 4</td>
<td>$ 0 a 4</td>
<td>+ b + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ 0 E 2</td>
<td>+ b + c</td>
<td>reduce (3)</td>
<td>$ 0 E 1 + 3</td>
<td>b + c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>+ b + c</td>
<td>reduce (4)</td>
<td>$ 0 E 1 + 3 b 4</td>
<td>+ c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ 0 E 1 + 3 T 5 + c</td>
<td>reduce (2)</td>
<td>$ 0 E 1 + 3 T 5</td>
<td>+ c</td>
<td>reduce (2)</td>
<td></td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>+ c</td>
<td>shift</td>
<td>$ 0 E 1 + 3</td>
<td>c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1 + 3 T 5</td>
<td>+ c</td>
<td>reduce (4)</td>
<td>$ 0 E 1 + 3 T 5</td>
<td>+ c</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>+ c</td>
<td>reduce</td>
<td>$ 0 E 1</td>
<td>+</td>
<td>accept</td>
</tr>
</tbody>
</table>

Reference: Barbara Ryder’s 198:515 lecture notes
Syntax-directed Translation: AST Construction example

Grammar with production rules

```
S: E  { $S = $1; }  
E: E '+' T  { $S = new node("+", $1, $3); }  
   | T  { $S = $1; }  
T: T_ID  { $S = 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_dec*  
   | {-> New program(cst_main_class, main_class, class_decl, class_decl) }

cst_exp_list -> exp*  
   | {-> many_rule }
   | {-> empty_rule }

cst_exp_rest -> exp*  
   | {-> t_comma cst_exp }

cst_type -> type  
   | {-> int_rule }
```

Abstract Syntax Tree

```
program =  
   | main_class [classdecls];classdecl*;  
   | {-> New int rule (int) | (bool) | ... }
```

AST, SymTable, IRT Tree, Assem, and MIPS for example

```
class And {
   public static void main(String[] a) {
      System.out.println(new Foo().testing(42));
   }
}
class Foo {
   public int testing(int p) {
      int x;
      if (p < 10 && 2 < p) {
         x = 7;
      } else {
         x = 22;
      }
      return x;
   }
}
```

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.

Parsing Terms
- see attached slides, be familiar with these terms
Next Time

Lecture
- More undergraduate compilers review

Parsing Terms

CFG (Context-free Grammar)
- production rule
- terminal
- nonterminal

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)