I. Syntax Analysis
   A. Breaking the program into logical units
      1. Input: token stream
      2. Output: representation of the logical units
         • Sequence of rules
         • Parse trees
         • Abstract syntax trees
   B. Also called parsing
      1. At one point, considered the core of compilers
   C. How to represent the logical units of a program
      1. Typically done in terms of a grammar
      2. Typically done in terms of a context free grammar
      3. This is what is found in the language definition, language manual, etc.
   D. Examples
      1. Various books, A68

II. Formal model
   A. Issue: is a string of tokens a valid program
      1. This is a language issue
      2. Lexical analysis: is this string of characters a valid token
   B. Context free grammars
      1. Regular grammars (FSAs) are inappropriate for parsing
         • WHY
      2. CFGs work pretty well for most language
         • What does pretty well mean – are languages really context free?
         • Even if you can’t parse the language, you can parse a superset of the language
            i. How does this help
   C. Notation
      1. CFG = <Σ,N,T,S>
• Σ is the set of rules
• N is the set of nonterminals
• T is the set of terminal
• S is the start state

2. Rules are N := <sequence of terminals and nonterminals>
   • In general can have a regex of N,T on the right hand side
   • BNF (backus-normal, backus-naur form)
     i  A := { B }
     ii A := [ B ]
     iii A := B | C

D. Example grammar
   program ::= LX_PROGRAM LX_ID (' id_list ') ';' decls cmpdstmt
   id_list ::= LX_ID { ',' LX_ID }
   decls ::= { LX_VAR id_list ':' type ';' }  
   type ::= LX_INTEGER | LX_REAL
   cmpdstmt ::= LX_BEGIN stmt { ';' stmt } LX_END
   stmt ::= LX_ID LX_ASSIGNOP expr
   expr ::= LX_CONSTANT

E. Example source
   PROGRAM t (i,o);
   VAR x : INTEGER;
   BEGIN x := 1 END

F. Expand the grammar to get unique rules
   1. id_list ::= LX_ID id_list1
   2. id_list1 ::= 
   3. id_list1 ::= ',' LX_ID id_list1
   4. decls ::= 
   5. decls ::= LX_VAR id_list ':' type '; ' decls
   6. cmpdstmt ::= LX_BEGIN stmt_list LX_END
   7. stmt_list ::= stmt stmt_list1
   8. stmt_list1 ::= 
   9. stmt_list1 ::= ';' stmt

III. Context Free Grammars
   A. Derivations
      1. We generate a string by apply a sequence of rules
      2. Example
      3. This is called a derivation
4. Parsing is the opposite task

B. Representing derivations
   1. Sequence of rules is ambiguous
      • Assume left-most or right-most application of the next rule
   2. Trees are an unambiguous alternative
      • Pure syntax trees
      • Abstract syntax trees
   3. Example

C. CFGs are useful for defining languages
   1. This is what is used in reference manuals
   2. Can be syntactic, graphical

D. But CFGs can’t do everything
   1. Check that a variable is declared
   2. Check if an identifier is declared as a type name
      • Which might be needed to be form the grammar

E. CFGs are often ambiguous
   1. Easy to do by mistake
      • $\text{var} ::= \text{id} | \text{id}\![\text{expr}]$
      • $\text{expr} ::= \text{var} | \text{var}\![\text{expr}]$
   2. Easy to do on purpose
      • $\text{e} ::= \text{e} + \text{e} | \text{e} * \text{e} | a$
      • $\text{stmt} ::= \text{IF expr THEN stmt [ ELSE stmt ]}$
   3. Some CF languages are inherently ambiguous
      • Otherwise $n^2$ parsing would be possible
   4. Some programming languages are as well
      • $x(*y)$ in C
      • 0-ary function call in Pascal
      • $x(y)$ in Ada
   5. Ways around this
      • Can change grammar
         i. Expression grammar using levels (write out)
      • Can parse a superset of the language
         i. Disambiguate during semantic analysis
      • Put some smarts into lexical analysis
         i. Mini symbol table and LX_TYPEID tokens
         ii. cfront parsing lexical analysis
• This is why the grammar is typically designed first, then the set of tokens

IV. Top-Down Parsing
   A. This is the simplest approach
      1. Most intuitive at first
      2. Also the earliest approach historically
   B. Basic idea
      1. Work left to right (as in reading)
      2. Figure out what rule to apply next in a left-most derivation
      3. Build the parse tree top-down
   C. Example
   D. Problems
      1. Left recursion produces infinite loops
         • Replace with right recursion
      2. Look ahead might be needed
         • At least some times in a real language
         • Field versus method declaration
         • Declaration statement versus expression
         • Super call versus expression starting with super.
         • New of object versus new of array
         • Call versus name access
      3. Difficult (somewhat unnatural) to write expression grammar
         • E -> T { + T }
         • E -> T E1, E1 -> , E1-> + E
         • T -> F { * F }
   E. LL Parsing
      1. Top down parsing where you expand the first available nonterminal
         • Is call LL parsing. (Left to right scan of tokens, left-most derivation)
         • LLk – LL parsing with k token look-ahead
            i  LL(1), LL(2),...
            ii LL(k) – arbitrary look ahead
      2. In order to do LL(k) parsing, your grammar must be LL(k)
         • Must avoid these problems
      3. Take a grammar as given in the reference and translate to LL(k)
         • Generally easy
         • But not typically for expressions
      4. In general, LL(k) grammars are a proper subset of all grammars
• And a proper subset of unambiguous grammars as well

F. Writing a top-down parser
1. Recursive descent parser
   • Create a routine for each rule
   • Routine implements the RHS (regex) of the rule using code
     i. checkToken(TOK)
     ii. getToken() -> token, then switch
     iii. pushback
     iv. Similar to implementing regex for lexical analysis

G. Table-driven parser

<table>
<thead>
<tr>
<th>STATE</th>
<th>LOOKFOR</th>
<th>ACTION</th>
<th>GOTO</th>
<th>ALTERNATIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>LX_PROGRAM</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LX_ID</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>‘(‘</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Id_list</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘)’</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>‘;’</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>decls</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cmpdstmt</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LX_EOF</td>
<td>Recognize program</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Id_list</td>
<td>LX_ID</td>
<td>Create idlist w/ ID</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>‘;’</td>
<td></td>
<td>10</td>
<td>ALT:0</td>
</tr>
<tr>
<td>10</td>
<td>LX_ID</td>
<td>Add id to current idlist</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

V. Tools for top-down parsing
A. JAVACC is an example of a top-down parser generator
   1. Syntactic rules are BNF-style regular expressions
   2. Generate code as with recursive descent from the rules
   3. Rules can include java code fragments (actions) at appropriate places

B. What are the difficulties
   1. Suppose we have a grammar rule for a non terminal
      • Suppose there are alternatives (OR rule, multiple rules)
      • How do we choose among them
   2. Cases
      • Statement := if_statement | while_statement | expr_stmt ...
• declaration := field_decl | method_decl | class_decl

i Look at next token – it determines which
ii Need to know which token starts each alternative

• What is the problem

• How could you solve this
  ➢ Rewrite the grammar
    ➢ decl := modifiers decl1
    ➢ decl1 := LX_CLASS ...
        | type name decl2
    ➢ decl2 := ‘(‘ ...
        | [ ‘=’ expr ] ‘;’
  ➢ Do more than one character lookahead

C. Handling look ahead
1. Specify a lookahead value (# tokens), locally or globally
   • Example: ( A | B) where A and B can start with same token
     i super(…) versus super.x
     ii Need to look 2 tokens ahead
2. JAVACC lets you specify a constant lookahead at any point
   • Is this sufficient
     • Method versus constructor versus field declaration
       i All start with an arbitrarily long sequence of modifiers
3. JAVACC lets you specify a regular expression for lookahead
   • The rule is followed if the regular expression is matched
     i Then the tokens are pushed back and rescanned in the rule

D. Actions (doing something with the parse)
1. Each nonterminal is a routine
2. Routines can return values (declare them as part of grammar)
3. Routines can throw exceptions (error handling)
4. Code can be placed anywhere in the rule { ... }
   • Should follow LOOKAHEAD if that is needed
   • Won’t raise ambiguities otherwise
5. Code is arbitrary java code
   • Can use the value of a terminal/nonterminal (or assign for later)
   • Terminals are type Token, and Token.image yields the string
6. We’ll get back to this for the next phase

VI. Error Management
   A. Where can errors be detected in the compiler
1. Lexical analysis (unclosed strings, illegal characters)
2. Syntax analysis
3. Semantic analysis
4. Optimization
5. Code generation

B. What do you want to do with errors
   1. Tell the user in a meaningful way
   2. Recover sufficiently to provide more errors
   3. Recover sufficiently to correct the program
   4. Cease future processing

C. What are the problems here
   1. Need to provide locations in later stages
   2. Messages can come in random order
   3. How to recover appropriately
   4. How to avoid spurious errors

D. Error management class as part of the compiler
   1. Store errors, sort by line, print at end
   2. Track maximum error level (to know whether to continue to next phase)

VII. Error recovery techniques
A. Lexical analysis
   1. Don’t report errors, just pass on to the parser
      • Either as a special token (LX_ERROR)
      • Or as a token designed to elicit a specific error (LX_BADSTRING)
      • Or through an exception (JavaCC)
   2. The parser is better equipped to handle errors

B. Syntax analysis
   1. Detect syntactic errors
   2. Recover at logical locations

C. Semantic analysis
   1. Parsing needs to keep location information around for this
   2. Build error syntax trees for program analysis

VIII. Parser error management
A. Basic idea
   1. Parser will detect an error at the first point where there is not a valid continuation
      • This is inherent to both top-down and bottom-up parsing
      • Because we are working left to right
• The parser knows what tokens can follow
2. Generate an error message at that point
3. Restore the parser to a “known” state and continue
   • Find more than one error in the program
   • Avoid cascading errors
4. Known states
   • Easy to identify locations
   • Meaningful units for the programmer
   • Examples: statements, blocks, functions, classes
     i These have well-defined end and start
   • Parser can do look ahead to find such points
     i Should look for k tokens that match, not single point
     ii This is easier if the parser is automatically generated

B. JavaCC
1. Routines throw an exception on parse error
   • With some information as to what is there and what was expected
2. Routines for known states can catch the exception
   • Generate an error message
   • Do look ahead to find the end of rule
   • And then return
3. Can also add grammar rules for common errors

C. Error correction
1. It is possible to correct some errors using tables and look-ahead
   • You know what tokens are valid continuations
   • Consider possibilities
     i Inserting a missing token
     ii Deleting the next token (ignoring it)
     iii Replacing the next token with another
2. Why isn’t this used?

IX. Homework
1. Exercise 3.9