CS412/413

Introduction to Compilers
Radu Rugina

Lecture 10: Syntax-Directed Definitions
11 Feb 02

Parsing Techniques

- LL parsing
  - Computes a Leftmost derivation
  - Builds the derivation top-down
  - LL parsing table indicates which production to use for expanding the rightmost non-terminal
- LR parsing
  - Computes a Rightmost derivation
  - Builds the derivation bottom-up
  - Uses a set of LR states and a stack of symbols
  - LR parsing table indicates, for each state, what action to perform (shift/reduce) and what state to go to next
- Use these techniques to construct an AST

AST Review

- Derivation = sequence of applied productions
  \[ S \rightarrow E + S \rightarrow I + S \rightarrow I + E \rightarrow I + 2 \]
- Parse tree = graph representation of a derivation
  - Doesn’t capture the order of applying the productions
- Abstract Syntax Tree (AST) discards unnecessary information from the parse tree

AST Data Structures

```java
abstract class Expr {
    // ...
}
class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) {
        left = L; right = R;
    }
}
class Num extends Expr {
    int value;
    Num(int v) { value = v; }
}
```

Implicit AST Construction

- LL/LR parsing techniques implicitly build the AST

  - The parse tree is captured in the derivation
    - LL parsing: AST is implicitly represented by the sequence of applied productions
    - LR parsing: AST is implicitly represented by the sequence of applied reductions

  - We want to **explicitly** construct the AST during the parsing phase:
    - add code in the parser to explicitly build the AST

AST Construction

- LL parsing: extend procedures for non-terminals
- Example:

  ```java
  void parse_SR() {
      switch (token) {
      case num: case “”:
          parse_SR();
          parse_SR();
          return;
      default:
          throw new ParseError();
      }
  }
  ```

  ```java
  Expr parse_SR() {
      switch (token) {
      case num: case “":
          Expr left = parse_SR();
          Expr right = parse_SR();
          if (right == null) return new Num();
          else return new Add(left, right);
      default:
          throw new ParseError();
      }
  }
  ```
**AST Construction**

- LR parsing
  - We need again to add code for explicit AST construction

- AST construction mechanism for LR Parsing
  - Store parts of the tree on the stack
  - For each nonterminal symbol X on stack, also store the sub-tree rooted at X on stack
  - Whenever the parser performs a reduce operation for a production X → γ, create an AST node for X

**AST Construction for LR Parsing**

- Example

```
S → E + S | S
E → num | ( S )
```

**Problems**

- Unstructured code: mixed parsing code with AST construction code

- Automatic parser generators
  - The generated parser needs to contain AST construction code
  - How to construct a customized AST data structure using an automatic parser generator?

- May want to perform other actions concurrently with the parsing phase
  - E.g. semantic checks
  - This can reduce the number of compiler passes

**Syntax-Directed Definition**

- Solution: syntax-directed definition
  - Extends each grammar production with an associated semantic action (code):

```
S → E + S { action }
```

- The parser generator adds these actions into the generated parser
- Each action is executed when the corresponding production is reduced

**Semantic Actions**

- Actions = code in a programming language
  - Same language as the automatically generated parser

- Examples:
  - Yacc = write actions in C
  - CUP = write actions in Java

- The actions access the parser stack!
  - Parser generators extend the stack of symbols with entries for user-defined structures (e.g. parse trees)

- The action code should be able to refer to the grammar symbols in the production
  - Need a naming scheme...

**Naming Scheme**

- Need special names for grammar symbols to use in the semantic action code

- Need to refer to multiple occurrences of the same nonterminal symbol

```
E → E₁ + E₂
```

- Distinguish the nonterminal on the LHS

```
E₀ → E + E
```
Naming Scheme: CUP
• CUP:
  – Rename nonterminals using distinct, user-defined names:
    \[ \text{expr} ::= \text{expr1 PLUS expr2} \]
  – Use keyword RESULT for LHS nonterminal
• CUP Example:
  \[ \text{expr} ::= \text{expr1 PLUS expr2} \]
  \{ \text{RESULT} = \text{expr1} + \text{expr2}; \}

Naming Scheme: yacc
• Yacc:
  – Uses keywords: $1$ refers to the first RHS symbol, $2$ refers to the second RHS symbol, etc.
  – Keyword $$ refers to the LHS nonterminal
• Yacc Example:
  \[ \text{expr} ::= \text{PLUS expr} \]
  \{ $$ = $1 + $3; \}

Building the AST
• Use semantic actions to build the AST
• AST is built bottom-up along with parsing

Example
\[ E \rightarrow \text{num} | (E) | E + E | E * E \]
• Parser stack stores value of each non-terminal
  \[ (1+2)^3 \]
  \[ (E) \]

AST Design
• Keep the AST abstract
• Do not introduce a tree node for every node in parse tree (not very abstract)

AST Design
• Do not use one single class AST_node
• E.g. need information for if, while, +, *, ID, NUM
  class AST_node {
    int node_type;
    AST_node[ ] children;
    String name; int value; ...etc...
  }
• Problem: must have fields for every different kind of node with attributes
• Not extensible, Java type checking no help
Use Class Hierarchy

- Can use subdassing to solve problem
  - Use an abstract class for each "interesting" set of non-terminals in grammar (e.g. expressions)

\[ E \rightarrow E + E | E * E | E \] (E)  
abstract class Expr { ... } 
addd class Add extends Expr { Expr left, right; } 
abstract class Expr { OperOp, Expr l, r; } 
// or: class BinExpr extends Expr { OperOp; Expr l, r; }
abstract class Minus extends Expr { Expr e; } 

Another Example

\[ E ::= num | (E) | E + E | \text{id} \] 
\[ S ::= E | \text{if}(E) S | \text{if}(E) \text{else} S | \text{id} = E ; \] 
abstract class Expr {...} 
addd class Num extends Expr { num(int value) ... } 
addd class Add extends Expr { Add(Expr e1, Expr e2) ... } 
addd class Id extends Expr { Id(String name) ... } 
abstract class Stmt {...} 
addd class ExpExpr extends Stmt { ExpExpr e, Stmt stmt1, Stmt stmt2} 
addd class EmptyStmt extends Stmt { EmptyStmt ... } 
addd class AssignStmt extends Stmt { AssignStmt(String id, Exp e) ... } 

Other Syntax-Directed Definitions

- Can use syntax-directed definitions to perform semantic checks during parsing
  - E.g. type-checking
- Benefit = efficiency
  - One single compiler pass for multiple tasks
- Disadvantage = unstructured code
  - Mixes parsing and semantic checking phases
  - Perform checks while AST is changing

Type Declaration Example

\[ D \rightarrow T \text{id} \] 
\{ AddType(id, T.type); \} 
\[ D \rightarrow D_1 , \text{id} \] 
\{ AddType(id, D1.type); \} 
\{ D.type = D1.type; \} 
\[ T \rightarrow \text{int} \] 
\{ T.type = intType; \} 
\[ T \rightarrow \text{float} \] 
\{ T.type = floatType; \} 

Propagation of Values

- Propagate type attributes while building the AST

\[ \text{int a, b} \] 

Another Example

\[ D \rightarrow T \text{L} \] 
\{ AddType(id, T.type); \} 
\{ D.type = T.type; \} 
\{ L.type = D.type; \} 
\[ T \rightarrow \text{int} \] 
\{ T.type = intType; \} 
\[ T \rightarrow \text{float} \] 
\{ T.type = floatType; \} 
\[ L \rightarrow L_1 , \text{id} \] 
\{ AddType(id, L1.type); \} 
\{ ??? \} 
\[ L \rightarrow \text{id} \] 
\{ AddType(id, ???); \} 

4
### Propagation of Values

- Propagate values both bottom-up and top-down

```latex
define int a, b

[Diag: Type tree with labels]
```

- LR parsing: AST is built bottom-up

```latex
AddType(id, Type)
```

### Structured Approach

- Separate AST construction from semantic checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone
  - It is better when efficiency is not a critical issue

### Summary

- Syntax-directed definitions attach semantic actions to grammar productions
- Easy to construct the AST using syntax-directed definitions
- Can use syntax-directed definitions to perform semantic checks
- Separate AST construction from semantic checks or other actions which traverse the AST