CS412/413
Introduction to Compilers
Radu Rugina

Lecture 10: Syntax-Directed Definitions
16 Feb 04

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
- **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

![Parse Tree Diagram]

AST Data Structures

```java
abstract class Exp ( )

class Add extends Exp {
    Exp left, right;
    Add(Exp L, Exp R) {
        left = L; right = R;
    }
}

class Num extends Exp {
    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 nonterminals
- **Example**:

```java
void parse_S ( ) {
    switch (token) {
        case num: case ‘(‘:
            parse_E();
            return;
        default:
            throw new ParseError();
    }
}
```

```
void parse_E ( ) {
    switch (token) {
        case num: case ‘(‘:
            Expr left = parse_E();
            Expr right = parse_S();
            if (right == null) return left;
            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 \( X \rightarrow \gamma \), create an AST node for \( X \)

**AST Construction for LR Parsing**

- Example

```
S   +   E
    Num(1) Num(2) Num(3)
```

Before reduction
\( S \rightarrow E + S \)

After reduction
\( S \rightarrow E + S \)

**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 \rightarrow 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 \rightarrow E_1 + E_2
```

- Distinguish the nonterminal on the LHS

```
E_0 \rightarrow E + E
```
Naming Scheme: CUP

- CUP:
  - Rename nonterminals using distinct, user-defined names:
    \[ \text{expr ::= expr} \text{ PLUS expr} \text{ PLUS expr} \]
  - Use keyword RESULT for LHS nonterminal

- CUP Example:
  \[
  \text{expr ::= expr} \text{ PLUS expr} \text{ PLUS expr} \\
  \{ \text{RESULT = e1 + e2; \} } \]

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 ::= expr PLUS expr} \\
  \{ $$ = \text{$1 +$3; } \}
  \]

Building the AST

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

Example

\[
E \rightarrow \text{num | (E) | E} + \text{E} | E \text{ * E}
\]

- Parser stack stores value of each non-terminal:
  \[
  (1+2)*3 \\
  (E \text{ Num(1)} +2)*3 \text{ RESULT = new Num(1)} \\
  (E+2) \text{ Num(2)} +)*3 \text{ RESULT = new Num(2)} \\
  (E+\text{ Add+3}) \text{ Add +3} \text{ RESULT = new Add(e1,e2)} \\
  \text{E} \text{ Add +3} \text{ RESULT = e}
  \]

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 {
  \text{int node_type;}
  \text{AST_node[ ] children;}
  \text{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 subclassing to solve problem
  - Use an abstract class for each "interesting" set of non-terminals in grammar (e.g., expressions)

```
E ::= E + E | E * E | -E | (E)
abstract class Expr {...}
class Add extends Expr { Expr left, right; ...}
class Mult extends Expr { Expr left, right; ...}
// or: class BinExpr extends Expr { Oper o; Expr l, r; }
class Minus extends Expr { Expr e; ...}
```

Another Example

```
E ::= num | (E) | E + E | id
S ::= E ; | if (E) S | if (E) S else S | id = E ; |

abstract class Expr {...}
class Num extends Expr { Num(int value) ...}
class Add extends Expr { Add(Expr e1, Expr e2) ...}
class Id extends Expr { Id(String name) ...}

abstract class Stmt {...}
class IfStmt extends Stmt { IfStmt(Expr c, Stmt s1, Stmt s2) }
class EmptyStmt extends Stmt { EmptyStmt ...}
class AssignStmt extends Stmt { AssignStmt(String id, Expr 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 → T id
    { AddType(id, T.type);
      D.type = T.type; }

D → D1 , id
    { AddType(id, D1.type);
      D.type = D1.type; }

T → int
    { T.type = intType; }

T → float
    { T.type = floatType; }
```

Propagation of Values

- Propagate type attributes while building the AST

```
int a, b
```

Another Example

```
D → T L
    { D.type = T.type;
      L.type = T.type; }

t → int
    { T.type = intType; }

t → float
    { T.type = floatType; }

L → id
    { AddType(id, ???); }

L → L1 , id
    { AddType(id, L1.type);
      ??? }
```
**Propogation of Values**

- Propagate values both bottom-up and top-down

```
Propogation of Values
```

```
int a, b
```

```
T.type
intType
```

```
L.type
```

```
D
```

```
AddType(id,L.type)
```

```
LR parsing: AST is built bottom-up!
```

```
AddType(id,L.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