CS42/413
Introduction to Compilers
Radu Rugina

Lecture 10: Syntax-Directed Definitions
10 Feb 03

---

**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
  - $S \rightarrow E + S \rightarrow 1 + S \rightarrow 1 + E \rightarrow 1 + 2$
- **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 nonterminals
- **Example**:

```java
void parse_S() {
    switch (token) {
        case num: case ":
            parse_S();
            parse_S();
            break;
        case expr: 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

---

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):
    
    \[ S \rightarrow E + S \quad \{ \text{action} \} \]

  - 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:
    
    ```
    expr ::= expr1 PLUS expr2
    { RESULT = expr1 + expr2;
    }
    ```
  - Use keyword RESULT for LHS nonterminal

- **CUP Example:**
  
  ```
  expr ::= expr1 PLUS expr2
  { RESULT = expr1 + 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:**
  
  ```
  expr ::= expr PLUS expr  { $$ = $1 + $3;
  }
  ```

Building the AST

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

![Diagram of AST building](Diagram)

- **Non terminal Expr:**
  - User-defined type for objects on the stack
  - Nonterminal name

  ```
  expr ::= NUM:
  { RESULT = new Num(l.val); }
  expr ::= expr1 PLUS expr2
  { RESULT = new Add(expr1,expr2); }
  expr ::= expr1 MULT expr2
  { RESULT = new Mul(expr1,expr2); }
  expr ::= LPAR expr RPAR
  { RESULT = e; }
  ```

Example

- **Parser stack stores value of each non-terminal**
  - 
  ```
  E → num | ( E ) | E + E | E * E
  ```

  - (1 + 2)*3
  - (E + 2) * 3
  - (E * E) * 3
  - (E + Add + 7) * 3
  - E * 3

- **RESULT = new Num(1)**
- **RESULT = new Add(expr1,expr2)**
- **RESULT = new Add(expr1,expr2)**
- **RESULT = new E**

AST Design

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

![Diagram of AST design](Diagram)

- **Problem:**
  - Must have fields for every different kind of node with attributes
  - Not extensible, Java type checking no help

- **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 subclasing 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 { ... }

class Add extends Expr { Expr left, right; ... }

class Mult extends Expr { Expr left, right; ... }

// or: class BinExpr extends Expr { Opar o; Expr l, r; }

class Minus extends Expr { Expr e; ... }

Another Example

\[
E ::= \text{num} | (E) | E + E | \text{id}
\]

\[
S ::= E ; | \text{if}(E) S | \text{if}(E) S \text{else} S | \text{id} = E ; |
\]

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 If extends Stmt { If(Expr c, Stmt s1, Stmt s2) }

class Empty extends Stmt { Empty() ... }

class Assign extends Stmt { Assign(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

\[
D \rightarrow T \text{id} \{ \text{AddType(id, T.type);}
\]

\[
D \rightarrow D_1 , \text{id} \{ \text{AddType(id, D_1.type);} \}
\]

\[
T \rightarrow \text{int} \{ T\text{.type} = \text{intType}; \}
\]

\[
T \rightarrow \text{float} \{ T\text{.type} = \text{floatType}; \}
\]

Propagation of Values

- Propagate type attributes while building the AST

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

\[
\text{D.type}
\]

\[
\text{AddType(id, D.type)}
\]

\[
\text{T.type}
\]

\[
\text{AddType(id, T.type)}
\]

\[
\text{intType}
\]

\[
\text{id}
\]

Another Example

\[
D \rightarrow T \text{L} \{ D\text{.type} = T\text{.type};
\]

\[
L\text{.type} = T\text{.type}; \}
\]

\[
T \rightarrow \text{int} \{ T\text{.type} = \text{intType}; \}
\]

\[
T \rightarrow \text{float} \{ T\text{.type} = \text{floatType}; \}
\]

\[
L \rightarrow \text{id} \{ \text{AddType(id, ???)}; \}
\]

\[
L \rightarrow L_1 , \text{id} \{ \text{AddType(id, L_1.type);} \}
\]

\[
???
\]
Propagation of Values

- Propagate values both bottom-up and top-down

```
int a, b
```

- 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