CS412/413
Introduction to Compilers
Radu Rugina

Lecture 10: AST Construction
13 Feb 06

AST Review
• Derivation = sequence of applied productions
  $ \Rightarrow E \Rightarrow 1 + E \Rightarrow 1 + 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

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 productions being applied
  – LR parsing: AST is implicitly represented by the sequence of reduction operations
• We want to explicitly construct the AST as we parse the input stream of tokens:
  – add code in the parser to explicitly build the AST

AST Data Structures

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; }
    }

LL Parsing and AST Construction

Expr parse_E() {
    switch(token) {
    case num: // E -> num
        Expr result = Num (token.value);
        token = input.read(); return result;
    case '+': // E -> E + S
        Expr result = parse_E();
        Expr result = parse_S();
        if (token != '+') throw new ParseError();
        token = input.read(); return result;
    default: throw new ParseError();
    }
LL Parsing and AST Construction

```java
Expr parse_E() {
    switch (token) {
        case '(':
            Expr left = parse_E();
            Expr right = parse_E();
            if (right == null) return left;
            else return new Add(left, right);
        default: throw new ParseError();
    }
}
```

LR Parsing and 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 B on stack
  - Whenever the parser performs a reduce operation for a production B → γ, create an AST node for B

LR Parsing and AST Construction

- **Example**

  ![LR Parsing and AST Construction Diagram](image)

  **Before reduction**
  - S → E+S

  **After reduction**
  - S → E+S

Problems

- **Hand-written parsers**
  - Mix parsing code with AST construction code by hand
  - Makes the parser difficult to maintain

- **Automatic parser generators**
  - Generated parser must 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 — actions written in C
  - CUP — actions written in Java

- **The actions access the parser stack**
  - Parser generators extend the stack of states (corresponding to RHS symbols) symbols with entries for user-defined structures (e.g., parse trees)
  - Need a naming scheme to refer to values on the parsing stack
Naming Scheme

• Need to refer to values of grammar symbols in the semantic action code
• Multiple occurrences of the same nonterminal symbol
  \[ E \to E_1 + E_2 \]
• Distinguish the nonterminal on the LHS
  \[ E_2 \to E + E \]

Naming Scheme: CUP

• CUP:
  – Name RHS nonterminal occurrences using distinct, user-defined labels:
    \[ \text{expr} ::= \text{expr}:1 \text{ PLUS } \text{expr}:2 \]
  – Use keyword RESULT for LHS nonterminal
• CUP Example:
  \[
  \begin{align*}
  \text{expr} & ::= \text{expr}:1 \text{ PLUS } \text{expr}:2 \\
  \{ \text{RESULT} = \text{new Add}(:1,:2); \}
  \end{align*}
  \]

Naming Scheme: yacc

• Yacc:
  – Uses keywords: $1$ refers to the first RHS symbol, $2$ refers to the second RHS symbol, etc.
  – Keyword $3$ refers to the LHS nonterminal
• Yacc Example:
  \[
  \begin{align*}
  \text{expr} & ::= \text{expr} \text{ PLUS } \text{expr} \\
  \{ \text{RESULT} = \text{new Add}(:1,:3); \}
  \end{align*}
  \]

Building the AST

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

Example

\[
E \to \text{num} \mid (E) \mid E+E \mid E*E
\]
• Parser stack stores value of each nonterminal

\[
\begin{align*}
1 & \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \\
2 & \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \\
\text{Num1} & \quad \text{Num1} \quad \text{Num1} \quad \text{Num1} \quad \text{Num1} \quad \text{Num1} \\
\text{E+} & \quad \text{E+} \quad \text{E+} \quad \text{E+} \quad \text{E+} \quad \text{E+} \\
\text{E*} & \quad \text{E*} \quad \text{E*} \quad \text{E*} \quad \text{E*} \quad \text{E*} \\
\text{add} & \quad \text{add} \quad \text{add} \quad \text{add} \quad \text{add} \quad \text{add} \\
\text{E} & \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E} \\
\text{RESULT} & \quad \text{RESULT} \quad \text{RESULT} \quad \text{RESULT} \quad \text{RESULT} \quad \text{RESULT}
\end{align*}
\]

AST Design

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

[Diagram of AST tree with nodes labeled with operations and values]
AST Design

- Do not use one single class AST node
- E.g., need information for if, while, +, *, ID, NUM

```java
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 sub-classing to solve problem
  - Use an abstract class for each "interesting" set of non-terminals in grammar (e.g. expressions)
    
    $$E \rightarrow E + E \mid E * E \mid -E \mid (E)$$

```java
abstract class Expr { ... }
class Add extends Expr { Add(Expr e1, Expr e2) ... }
class Multiply extends Expr { Mul(Expr e1, Expr e2) ... }
```

Another Example

```latex
\begin{align*}
E &::= \text{num} \mid (E) \mid E + E \mid \text{id} \\
S &::= \text{id} = E \mid \text{if} (E) S \\
& \mid \text{if} (E) S \text{else} S \mid \text{id} = E \\
\end{align*}
```

```java
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) ... }
```

EBNF: Extended BNF Notation

- Extended Backus-Naur Form = grammar specification that borrows regular expression syntax
  *\*, +, ( ), ? operators (also [X] means X?)

```latex
S \rightarrow ES' \\
S' \rightarrow e \mid +S \\
S \rightarrow E (+E)^*
```

- EBNF version: no position on + associativity
- EBNF supported in some recent parser generators
  - e.g., JavaCC
- CUP, yacc don't support EBNF

Ambiguity

```latex
\begin{align*}
S &\rightarrow S + E \mid E \\
E &\rightarrow \text{num} \\
E &\rightarrow E + E \\
E &\rightarrow E + E
\end{align*}
```

What happens if we run this grammar through LALR construction?

Shift/Reduce Conflict

```latex
\begin{align*}
E &\rightarrow E + E \\
E &\rightarrow \text{num} \\
E &\rightarrow E + E \\
E &\rightarrow E + E \\
\end{align*}
```

```latex
\begin{align*}
\text{shift/reduce} &\rightarrow \text{shift: 1+2+3} \\
&\quad \text{reduce: (1+2)+3} \\
&\quad 1+2+3 \\
&\quad \sim
\end{align*}
```
Grammar in CUP

terminal PLUS, LPAREN...
non terminal E;
precedence left PLUS;

RULE: when shifting ‘+’ conflicts with
  reducing a production, choose reduce

E ::= E PLUS E
  | LPAREN E RPAREN
  | NUMBER ;

Precedence

• CUP can also handle operator precedence

E → E + E | T
T → T × T | num | ( E )

E → E + E | E × E
  | num | ( E )

Conflicts without Precedence

E → E + E | E × E
  | num | ( E )

RULE: for conflicts choose reduce if production symbol has
  higher precedence than shifted symbol; choose shift if vice-versa

Precedence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E | E TIMES E | ...

RULE: for conflicts choose reduce if production symbol has
  higher precedence than shifted symbol; choose shift if vice-versa

Summary

• Look-ahead information makes SLR(1), LALR(1),
  LR(1) grammars expressive
• Automatic parser generators support LALR(1)
  grammars
• Precedence, associativity declarations simplify
  writing grammars