CS412/413

Introduction to Compilers

Radu Rugina

Lecture 10: AST Construction

13 Feb 06

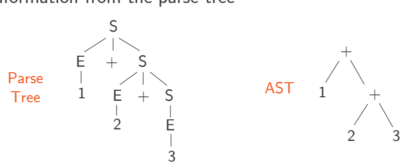
Parsing Techniques

- LL parsing
  - Computes a Leftmost derivation
  - Traverses the parse tree top-down
  - LL parsing table indicates which production to use for next
- LR parsing
  - Computes a Rightmost derivation
  - Traverses the parse tree bottom-up
  - Uses a parsing stack
  - Uses an LR parsing table that indicates what action to perform (shift/reduce) and what state to go to next
- The compiler also constructs an AST while parsing

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



Implicit AST Construction

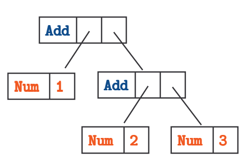
- LL/LR parsing techniques implicitly build the AST
- The parse tree is captured in the derivation
  - LL parsing: AST is implicitly represented by the sequence productions being applied
  - LR parsing: AST is implicitly represented by the sequence of reduction operations
- We want to explicitly construct 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

$S \rightarrow ES'$

$S' \rightarrow \varepsilon \mid +S$

$E \rightarrow \textbf{num} \mid (S)$

```
Expr parse_E() {
   switch(token) {
     case num: // E → number
          Expr result = Num (token.value);
          token = input.read(); return result;

     case '(':  // E → ( S )
          token = input.read();
          Expr result = parse_S();
          if (token != ')') throw new ParseError();
          token = input.read(); return result;

     default: throw new ParseError();
   }
}
```

LL Parsing and AST Construction

$S \to ES'$
$S' \to \varepsilon \mid +S$
$E \to \textbf{num} \mid (S)$

```
Expr parse_S() {
  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();
  }
}
```

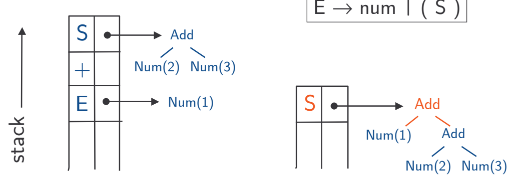

LR Parsing and AST Construction

- 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, also store the sub-tree rooted at B on stack
  - Whenever the parser performs a reduce operation for a production $B \to \gamma$, create an AST node for B

LR Parsing and AST Construction

- Example

$S \to E{+}S \mid S$
$E \to \text{num} \mid (S)$



Before reduction
$S \to E{+}S$

After reduction
$S \to E{+}S$

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):

$$S \to E{+}S \quad \{ \text{ action } \}$$

  - 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 = 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

$$E \rightarrow E_1 + E_2$$

- Distinguish the nonterminal on the LHS

$$E_0 \rightarrow E + E$$

Naming Scheme: CUP

- CUP:
  - Name RHS nonterminal occurrences using distinct, user-defined labels:

```
expr ::= expr:e1 PLUS expr:e2
```

  - Use keyword RESULT for LHS nonterminal

- CUP Example:

```
expr ::= expr:e1 PLUS expr:e2
        {:  RESULT = new Add(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:

```
expr ::= expr PLUS expr
           { $$ = $1 + $3; }
```


Building the AST

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

User-defined type for semantic objects on the stack

Nonterminal name

```
non terminal Expr expr;

expr ::= NUM:i                 {: RESULT = new Num(i.val); :}
expr ::= expr:e1 PLUS expr:e2  {: RESULT = new Add(e1,e2); :}
expr ::= expr:e1 MULT expr:e2  {: RESULT = new Mul(e1,e2); :}
expr ::= LPAR expr:e RPAR      {: RESULT = e; :}
```


Example

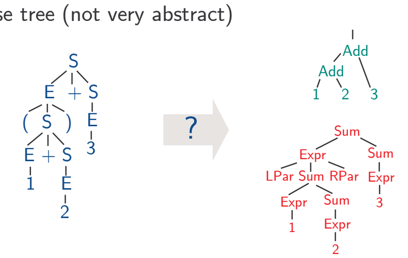
$$\boxed{E \rightarrow num \mid (E) \mid E+E \mid E*E}$$

- Parser stack stores value of each nonterminal

		(1+2)*3	
(1		+2)*3	
(E	Num(1)	+2)*3	RESULT=new Num(1)
(E+2		)*3	
(E+E	Num(2)	)*3	RESULT=new Num(2)
(E	Add(,)	)*3	RESULT=new Add(e1,e2)
(E)		*3	
E		*3	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 {
    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)

$$E \rightarrow E{+}E \mid E{*}E \mid {-}E \mid (E)$$

```
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 ::= id = 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 IfS extends Stmt { IfS(Expr c, Stmt s1, Stmt s2) }
class EmptyS extends Stmt { EmptyS() … }
class AssignS extends Stmt { AssignS(String id, Expr e)…}
```


EBNF: Extended BNF Notation

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

$$S \rightarrow ES' \qquad S' \rightarrow \varepsilon \mid +S \quad \Rightarrow \quad S \rightarrow E\,(+E)^*$$

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

Ambiguity

$$S \rightarrow S + E \mid E \qquad E \rightarrow num \quad \Rightarrow \quad E \rightarrow E + E \qquad E \rightarrow num$$

What happens if we run this grammar through LALR construction?

Shift/Reduce Conflict

E → E + E
E → num

E → E + E .	+
E → E . + E	+,$

$\xrightarrow{+}$

shift/reduce conflict

shift: 1+(2+3)
reduce: (1+2)+3

1+2+3
 ^

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

$E \to E + E \mid T$

$T \to T \times T \mid \text{num} \mid (E)$

$\Downarrow$

$E \to E + E \mid E \times E$

$\mid \text{num} \mid (E)$

Conflicts without Precedence

$E \to E + E \mid E \times E$

$\mid \text{num} \mid (E)$

$E \to E\,.+E$	...
$E \to E \times E\,.$	$+$

$E \to E + E\,.$	$\times$
$E \to E\,.\times E$	...

Predecence 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

$E \to E\,.+E$	...
$E \to E \times E\,.$	$+$

reduce $E \to E \times E$

$E \to E + E\,.$	$\times$
$E \to E\,.\times E$	...

Shift $\times$

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