CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 7: LL parsing and AST construction

February 4, 2008

LL(1) Parsing

- Last time:
  - how to build a parsing table for an LL(1) grammar (use FIRST/FOLLOW sets)
  - how to construct a recursive-descent parser from the parsing table

- Grammars may not be LL(1)
  - Use left factoring when grammar has multiple productions starting with the same symbol.
  - Other problematic cases?

if-then-else

- How to write a grammar for if stmts?

  $S \rightarrow$ if (E) S

  $S \rightarrow$ if (E) S else S

  $S \rightarrow$ other

Is this grammar ok?

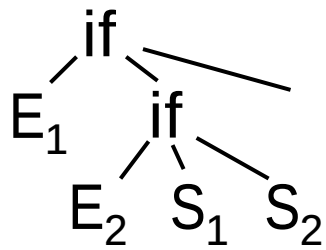
No—Ambiguous!

- How to parse?

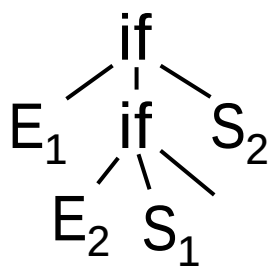
  if (E_1) if (E_2) S_1 else S_2

$S \rightarrow$ if (E) S
$S \rightarrow$ if (E) S else S
$S \rightarrow$ other

S $\rightarrow$ if (E) **S**
$\rightarrow$ if (E) if (E) S else S



S $\rightarrow$ if (E) **S** else S
$\rightarrow$ if (E) if (E) S else S



Which "if" is the "else" attached to?

Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Impose that unmatched "if" statements occur only on the "else" clauses

statement	→	matched | unmatched
matched	→	if (E) _____ else _____
	|	other
unmatched	→	if (E) _____
	|	if (E) _____ else _____

Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Impose that unmatched "if" statements occur only on the "else" clauses

statement	→	matched | unmatched
matched	→	if (E) matched else matched
	|	other
unmatched	→	if (E) _____
	|	if (E) _____ else _____

Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Impose that unmatched "if" statements occur only on the "else" clauses

statement	→	matched | unmatched
matched	→	if (E) matched else matched
	|	other
unmatched	→	if (E) _____
	|	if (E) matched else unmatched

Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Impose that unmatched "if" statements occur only on the "else" clauses

statement	→	matched | unmatched
matched	→	if (E) matched else matched
	|	other
unmatched	→	if (E) statement
	|	if (E) matched else unmatched

LL(1) if-then-else?

statement $\rightarrow$ if (E) ... | other

LL(1) if-then-else?

statement	$\rightarrow$	if (E) matched optional-tail | other
matched	$\rightarrow$	...
optional-tail	$\rightarrow$	else tail | ε
tail	$\rightarrow$	if (E) tail | other

LL(1) if-then-else?

statement	$\rightarrow$	if (E) matched optional-tail | other
matched	$\rightarrow$	if (E) matched else matched | other
optional-tail	$\rightarrow$	else tail | ε
tail	$\rightarrow$	if (E) tail | other

Left-Recursive Grammars

- Left-recursive grammars are not LL(1) !

$$S \rightarrow S\ \alpha$$
$$S \rightarrow \beta$$

- FIRST(β) $\subseteq$ FIRST(Sα)
- Both productions will appear in the predictive table, at row S in all the columns corresponding to symbols in FIRST(β)

Eliminate Left Recursion

- Method for left-recursion elimination:

  Replace

  $$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_m$$

  $$A \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

  with

  $$A \rightarrow \beta_1\ A' \mid \ldots \mid \beta_n\ A'$$

  $$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$$

- (See the complete algorithm in the Dragon Book)

Creating an LL(1) Grammar

- Start with a left-recursive grammar:

  $S \rightarrow S+E$

  $S \rightarrow E$

  and apply left-recursion elimination algorithm:

  $S \rightarrow ES'$

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

- Start with a right-recursive grammar:

  $S \rightarrow E+S$

  $S \rightarrow E$

  and apply left-factoring to eliminate common prefixes:

  $S \rightarrow E\ S'$

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

Top-Down Parsing

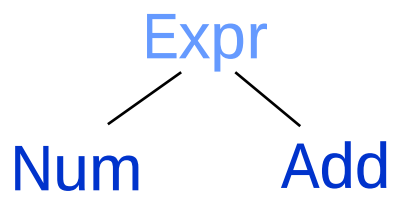
- Now we know:
  - how to build a parsing table for an LL(1) grammar (use FIRST/FOLLOW sets)
  - how to construct a recursive-descent parser from the parsing table

- Can we use recursive descent to build an abstract syntax tree too?

Creating the AST

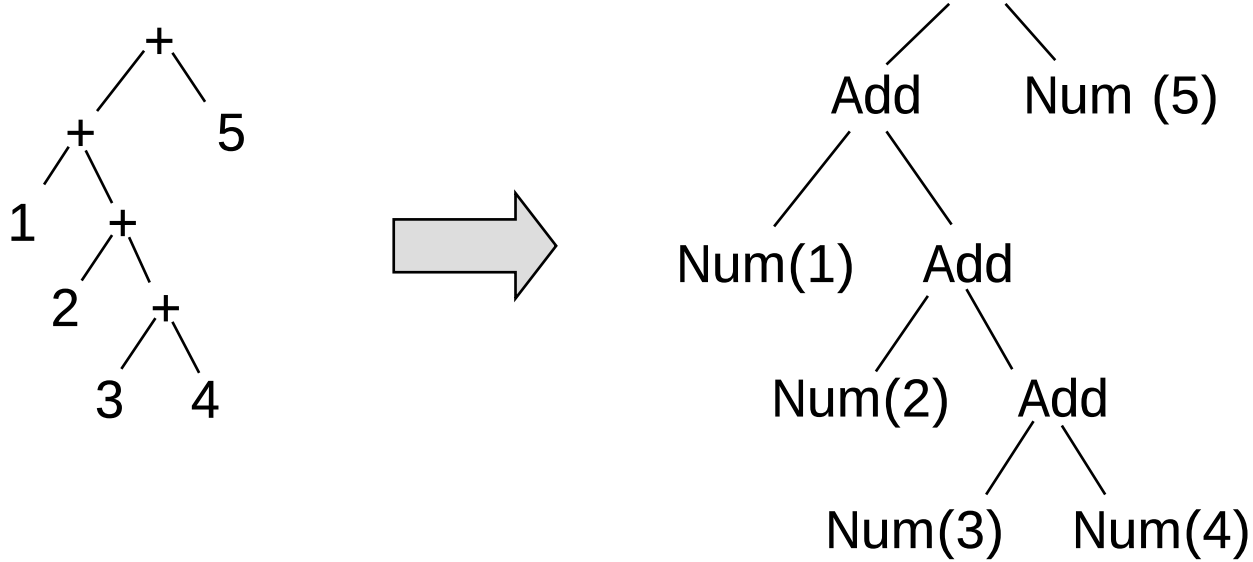
```
abstract class Expr { }

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; }
}
```

Class Hierarchy



AST Representation

(1 + 2 + (3 + 4)) + 5



How can we generate this structure during recursive-descent parsing?

Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- parse_S, parse_S', parse_E all return an Expr:

```
void parse_E()          Expr parse_E()
void parse_S()    ⇒     Expr parse_S()
void parse_S'()         Expr parse_S'()
```

AST Creation: parse_E

```
Expr parse_E() {
    switch(token) {
      case num:  // E → num
            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();
    }
}
```

AST Creation: parse_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();
    }
}
```

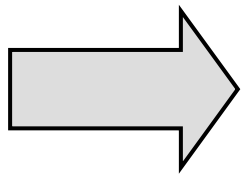
$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \textbf{num} \mid (S)$

EBNF: Extended BNF Notation

- Extended Backus-Naur Form = a form of specifying grammars which allows some regular expression syntax on RHS

  *, +, ( ), ? operators (also [X] means X?)

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



$S \rightarrow E(+E)^*$

- EBNF version: agnostic on + associativity

Top-down Parsing EBNF

- Recursive-descent code can directly implement the EBNF grammar:

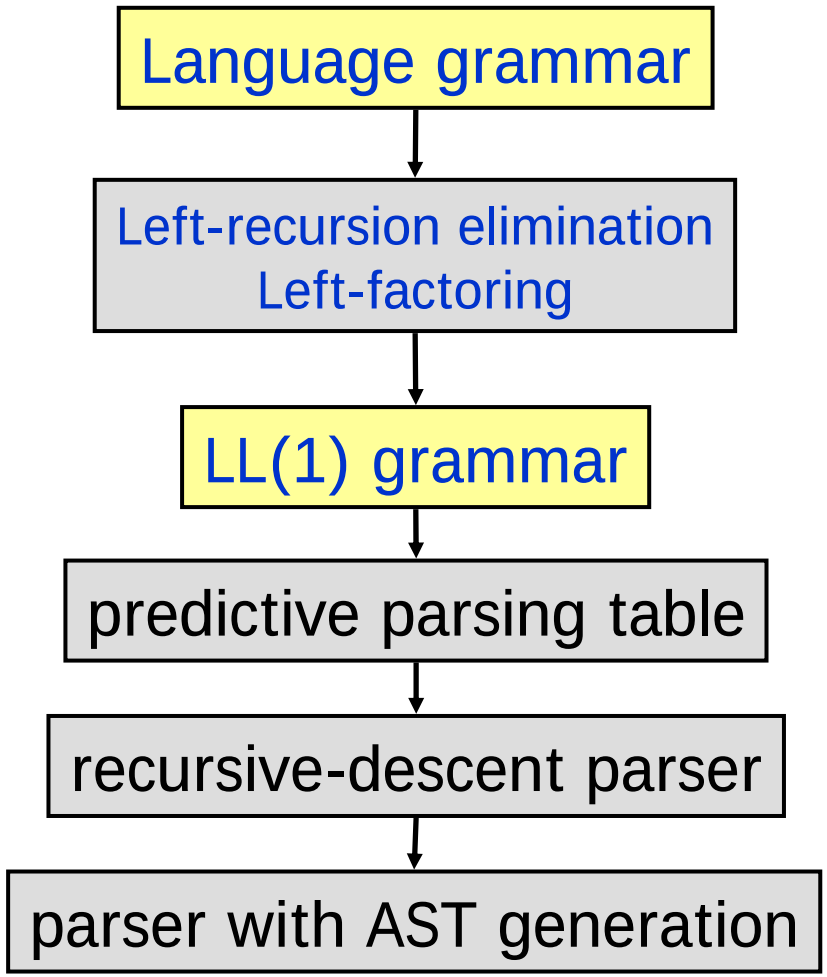
$$S \rightarrow E(+E)^*$$

```
void parse_S () { // parses sequence of E+E+E ...
    parse_E ();
    while (true) {
        switch (token) {
            case '+': token = input.read(); parse_E();
                      break;
            case ')': case EOF: return;
            default: throw new ParseError();
    }
}
```

Reassociating the AST

```
Expr parse_S() {
    Expr result = parse_E();
    while (true) {
      switch (token) {
        case '+': token = input.read();
                  result = new Add(result, parse_E());
                  break;
        case ')': case EOF: return result;
        default: throw new ParseError();
    }
}
```

Top-Down Parsing Summary



Exercises

- Which of the following are LL(1)?

(1)

$A \rightarrow aACd \mid b$

$C \rightarrow c \mid \varepsilon$

(2)

$A \rightarrow aACd \mid b$

$C \rightarrow A \mid \varepsilon$

(3)

$A \rightarrow aAC \mid b$

$C \rightarrow c \mid \varepsilon$