CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 7: LL parsing and AST construction
February 7, 2005

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 → if (E) S
S → if (E) S else S
S → other

Is this grammar ok?

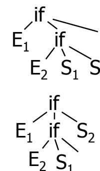
No—Ambiguous!

- How to parse?
  if (E_1) if (E_2) S_1 else S_2

S → if (E) S
S → if (E) S else S
S → other

S → if (E) **S**
→ if (E) if (E) S else S

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

LL(1) if-then-else?

statement	→ if (E) matched optional-tail | other
matched	→ if (E) matched else matched | other
optional-tail	→ else tail | ε
tail	→ if (E) tail | other

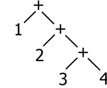
Left vs. Right Associativity

- Have been using grammar for language of "sums with parentheses" e.g., (1+(3+4))+5
- Started with simple, right-associative grammar:
  S → E + S | E
  E → num | (S)
- Transformed it to an LL(1) grammar by left-factoring:
  S → ES′
  S′ → ε | + S
  E → number | (S)
- What if we start with a left-associative grammar?
  S → S + E | E
  E → num | (S)

Left vs. Right Associativity

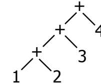
Right recursion : right-associative

S → E+S
S → E
E → num



Left recursion : left-associative

S → S+E
S → E
E → num



Left Recursion

- Left-recursive grammars don't work with top-down parsing: we don't know where to stop the recursion

derived string	lookahead	read/unread
S	1	1 + 2 + 3 + 4
S + E	1	1 + 2 + 3 + 4
S + E + E	1	1 + 2 + 3 + 4
S + E + E + E	1	1 + 2 + 3 + 4
E + E + E + E	1	1 + 2 + 3 + 4
1 + E + E + E	2	1 + 2 + 3 + 4
1 + 2 + E + E	3	1 + 2 + 3 + 4
1 + 2 + 3 + E	4	1 + 2 + 3 + 4
1 + 2 + 3 + 4	ε	1 + 2 + 3 + 4

Left-Recursive Grammars

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

  S → S α
  S → β
- FIRST(β) ⊆ 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

  $A \to A\ \alpha_1 \mid \dots \mid A\ \alpha_m$

  $A \to \beta_1 \mid \dots \mid \beta_n$

  with

  $A \to \beta_1\ A' \mid \dots \mid \beta_n\ A'$

  $A' \to \alpha_1\ A' \mid \dots \mid \alpha_m\ A' \mid \varepsilon$
- (See the complete algorithm in the Dragon Book)

Creating an LL(1) Grammar

- Start with a left-recursive grammar:
  S → S+E
  S → E
  and apply left-recursion elimination algorithm:
  S → ES′
  S′ → +E S′ | ε
- Start with a right-recursive grammar:
  S → E+S
  S → E
  and apply left-factoring to eliminate common prefixes:
  S → E S′
  S′ → + S | ε

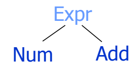
Top-Down Parsing

- 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

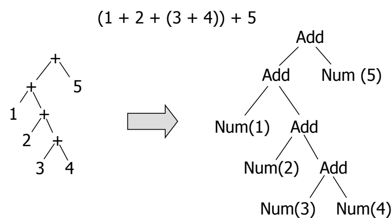
```
abstract class Expr { }

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



Add
Add Num (5)
Num(1) Add
Num(2) Add
Num(3) Num(4)

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 → 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();
    }
}
```


AST Creation: parse_S

$S \to ES'$
$S' \to \varepsilon \mid +S$
$E \to$ **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();
  }
}
```

Or…an Interpreter!

```
S → E S '
S ' → ε | +S
E → num | (S)
```

```
int parse_E() {
    switch(token) {
        case number:
            int result = token.value;
            token = input.read(); return result;
        case '(':
            token = input.read();
            int result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
        default: throw new ParseError(); }}

int parse_S() {
    switch (token) {
        case number:
        case '(':
          int left = parse_E();
          int right = parse_S();
          if (right == 0) return left;
          else return left + right;
        default: throw new ParseError(); } }
```


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?)

$S \to ES'$
$S' \to \varepsilon \mid +S$ ⇒ $S \to E(+E)^*$

- EBNF version: no position on + associativity

Top-down Parsing EBNF

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

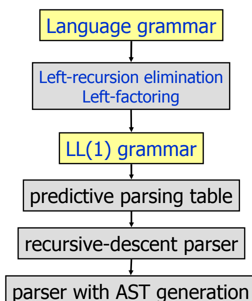
$S \to E(+E)^*$

```
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)
$A \to aACd \mid b$
$C \to c \mid \varepsilon$

(2)
$A \to aACd \mid b$
$C \to A \mid \varepsilon$

(3)
$A \to aAC \mid b$
$C \to c \mid \varepsilon$