COMP261 Lecture 19

Lindsay Groves

Parsing 4 of 4:

When does recursive descent work?

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Extending the language: Printing AST

NumberNode:
```
public String toString(){
    return String.format("%.5f", value);
}
```

AddNode:
```
public String toString(){
    String ans = "(" + first;
    for (Node nd : rest){ ans += " + "+ nd; }
    return ans + ")";
}
```

SubNode:
```
public String toString(){
    String ans = "(" + first;
    for (Node nd : rest){ ans += " - "+ nd; }
    return ans + ")";
}
```

Extending the language: Examples

Expr: add(10.5 ,-8)

Print → (10.5 + -8.0)

Value → 2.500

Expr: add(sub(10.5 ,-8), mul(div(45, 5), 6.8))

Print → ((10.5 – –8.0) + ((45.0 / 5.0) * 6.8))

Value → 79.700

Expr: add(14.0, sub(mul(div (1.0, 28), 17), mul(3, div(5, sub(7, 5)))))

Print → (14.0 + (((1.0 / 28.0) * 17.0) – (3.0 * (5.0 / (7.0 – 5.0)))))

Value → 7.107

Ex: Can you minimize the number or brackets used?

Extending the language: parser

`Allow add(1,2,3), etc.`

```
public Node parseAdd(Scanner s) {
   List<Node> args = new ArrayList<Node>();
   require(addPat, "Expecting add", s);
   require(openPat, "Missing '('", s);
   args.add(parseExpr(s));
   do {
      require (commaPat, "Missing ','", s);
      args.add(parseExpr(s));
    } while (!s.hasNext(closePat));
    require(closePat, "Missing ')'", s);
    return new AddNode(args);
 }
```

(need new version of require, taking a Pattern instead of a String)

Recursive Descent (LL(1)) Parsing - recap

- Method for each nonterminal/Node type
- Peek at next token to determine which branch to follow
- Build and return node
- Throw error (including error message) when parsing breaks
- Use require(…) to wrap up "check then consume/return or fail"
- Adjust grammar to make it cleaner
- LL(1) = deterministic, left-to-right, top down parsing with one symbol lookahead

Less Restricted Grammars

- When does this work?

- For example:

```
IfStmt ::= “if” “(“ Cond “)” Stmt  |
           “if” “(“ Cond “)” Stmt “else” Stmt
```

```
E ::= number |  E “+” E   |   E “–” E   |
      E “*” E   |   E  “/”  E
```

- What can we do about it?

When does it work?

- If we have a grammar rule:
  $N ::= W_1 \mid W_2 \mid \ldots \mid W_n$
  where W_i are sequences of terminal and/or nonterminal symbols.
- We must be able to tell which alternative to take, by looking just at the next input token.
- LL(1) condition 1: For any i and j ($i \neq j$) there is no symbol that can start both an instance of W_i and an instance of W_j.
- Easy to check if W_i and W_j start with terminals.
- What if they start with nonterminals?

What can we do? - Left-factoring

- Consider this grammar rule:

```
IfStmt ::= “if” “(“ Cond “)” Stmt  |
           “if” “(“ Cond “)” Stmt “else” Stmt
```

- If we see an “`if`”, we can't tell which branch to take.

- We can fix this by “factoring” out the common part:

```
IfStmt ::= “if” “(“ Cond “)” Stmt RestIf
RestIf ::= “” | “else” Stmt
```

Empty string

What can we do? - Left-factoring

- We can now parse this ok:
-
```
public Node parseIfStmt(Scanner s) {
  require(ifPat, “Missing ‘if’”, s);
  require(lbracPat, “Missing ‘(‘”, s);
  Node c = parseExp(s);
  require(lbracPat, “Missing ‘(‘”, s);
  Node thenPart = parseStmt(s);
  Node elsePart = parseRestIf(s);
  return new IfNode(c, thenPart, elsePart);
}
public Node parseRestIf(Scanner s) {
  if ( s.hasNext(elsePat) ) {
    s.next(); return parseStmt(s);
  } else { return null; }
```

Take the empty branch if no
other branch is possible.
Using null to represent empt

What can we do? - Left-factoring

- We can apply this idea to lots of grammars.

Assume nonterminals are upper case and terminals are lower cas

```
– A := B c | B d                A := B E
                                E ::= c | d

– A := B c | D e                A ::= h M | f g c | k l e
  B := f g | h i                M ::= i c | j e
  D := h j | k l
```

- These can be done using simple algebraic laws – like simplifying Boolean expressions
- For now, stick to basic grammars, and avoid using extensions like [...], [...]+ and [...]*.

When does it work?

- Consider the following grammar for lists of identifiers separated by commas.
- Informally, a list is either an identifier, or two lists separated by a comma.

L ::= *id* | L "," L

- This grammar is *ambiguous* – we can construct more that one parse tree for some strings.
- Ex: Draw all parse trees for "a,b" and "a,b,c".
- Recursive descent doesn't work for ambiguous grammars – must be able to construct a unique parse tree for any text in the language.

Left-recursion

- We can rewrite the grammar as:
  L ::= *id* | L "," *id*
- This is unambigous – draw parse trees as before.
- But … any L starts with an *id*.
- So, if we see an *id* we can' t tell which branch to take!
- In this case, we can' t factor out the common parts.
- Ex: try it!
- Recursive descent doesn't work for grammars with left-recursive rules (where the nonterminal on the left occurs at the start of some branch on the right).
- This breaks LL(1) condition 1!

Right-recursion

- We can also rewrite the grammar as:

  L ::= *id* | *id* "," L

- This is also unambigous – draw parse trees as before.
- And now we can factor out the common parts.

  L ::= *id* R (or L ::= *id* ["," L])
  R ::= "" | "," L

- What does this do to the parse trees? Does it matter?
- Ex: Write a parser for this grammar.

  Recall the way empty was handled in parsing If statements.

Infix expressions

- Consider the following grammar for arithmetic expressions:

E ::= *number* | E “+” E | E “–” E | E “*” E | E “/” E | “(“ E “)”

- This, again, is ambiguous – can get many different parse trees for some expressions.

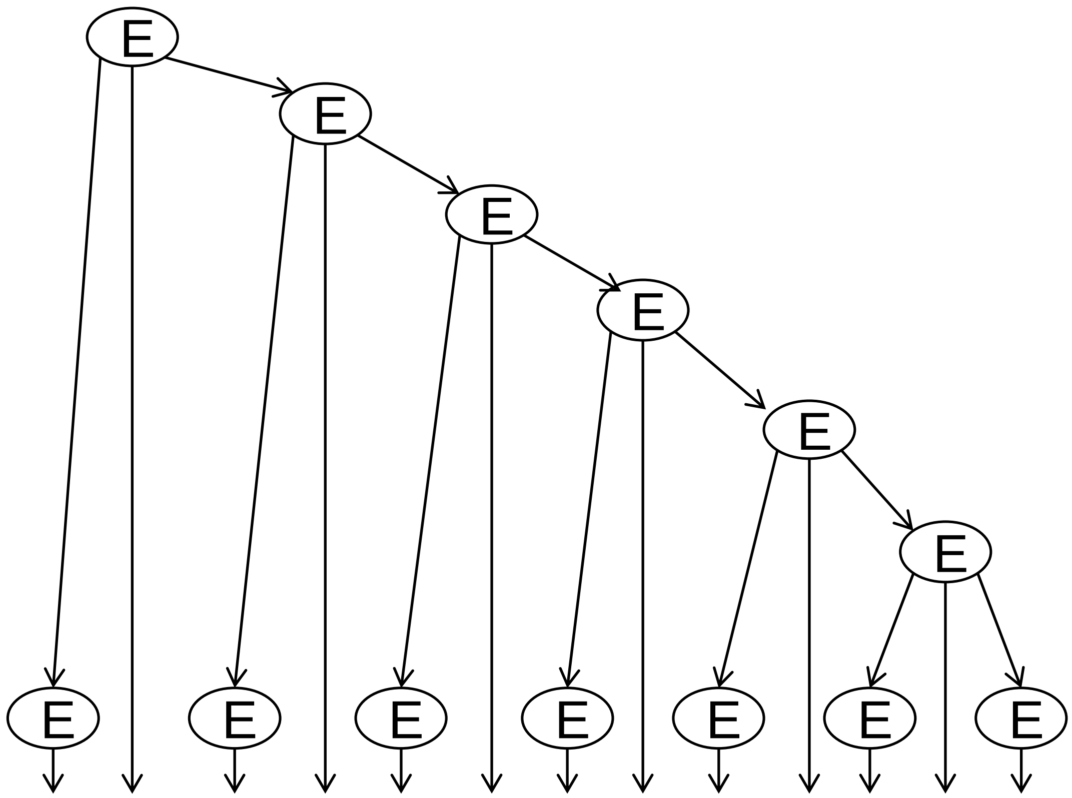
- Does it matter which parse tree we use?

- Think about order of evaluation!

Infix expressions

Grammar:

E ::= *number* | E "+" E | E "–" E | E "∗" E | E "/" E

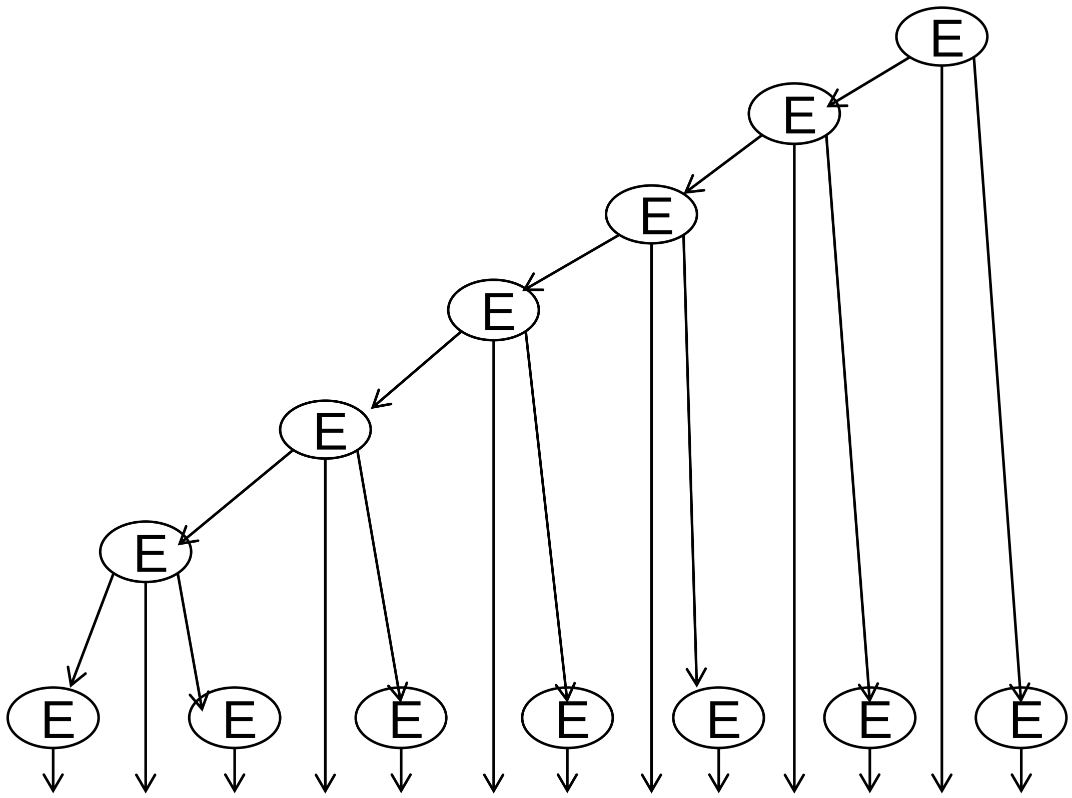


65 * 74 – 68 + 25 * 5 / 3 + 16

Infix expressions

Grammar:

E ::= *number* | E "+" E | E "–" E | E "∗" E | E "/" E

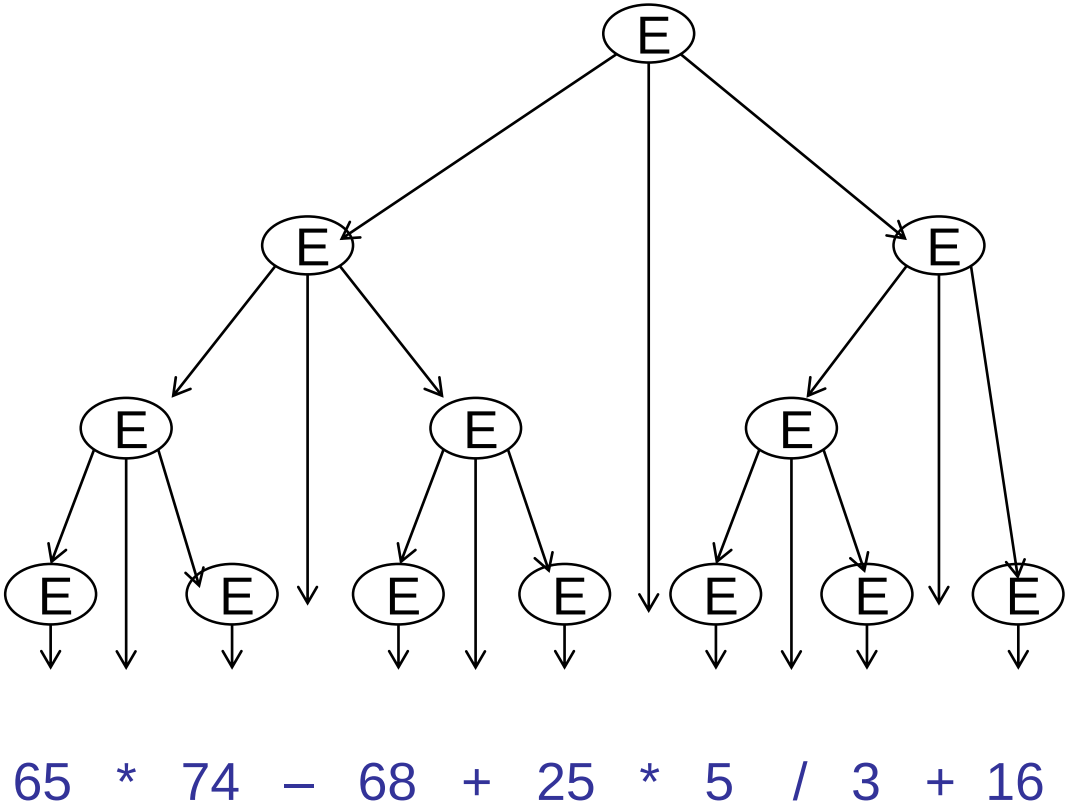


65 * 74 – 68 + 25 * 5 / 3 + 16

Infix expressions

Grammar:

E ::= *number* | E "+" E | E "–" E | E "∗" E | E "/" E



Infix Expressions

- We can make the grammar unambiguous by making it right-recursive, as for lists.

  EXPR ::= *number* | *number* "+" EXPR | *number* "–" EXPR |
  number "∗" EXPR | *number* "/" EXPR

- And the make it LL(1) by left-factoring:

  EXPR ::= *number* RESTOFEXPR

  RESTOFEXPR ::= "+" EXPR | "–" EXPR | "∗" TERM |
  "/" TERM | ""

- What does this do to the parse tree?
- Is that what we want?

Infix Expressions

- We can handle precedence by introducing an extra nonterminal.

  EXPR ::= TERM | TERM "+" EXPR | TERM "–" EXPR

  TERM ::= *number* | *number* "∗" TERM | *number* "/" TERM

- And the make it LL(1) by left-factoring:

  EXPR ::= TERM RESTOFEXPR

  RESTOFEXPR ::= "+" EXPR | "–" EXPR | ""

  TERM ::= *number* RESTOFTERM

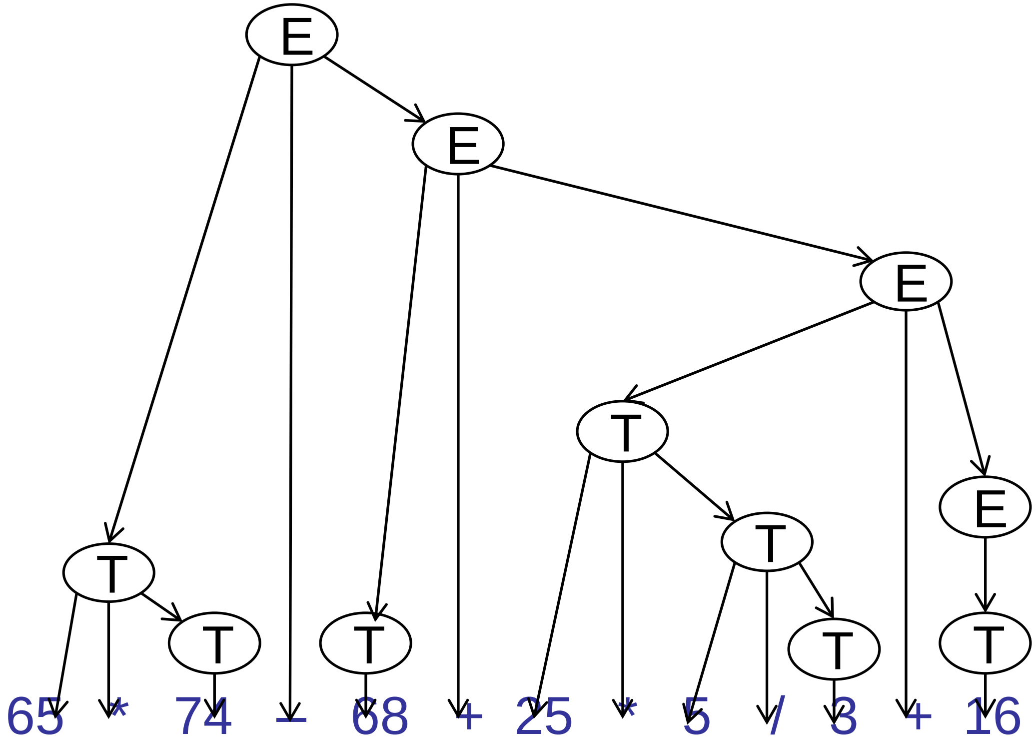
  RESTOFTERM ::= "∗" TERM | "/" TERM | ""

Infix Expressions

- What does this do to the parse tree?

  EXPR ::= TERM | TERM "+" EXPR | TERM "–" EXPR

  TERM ::= *number* | *number* "∗" TERM | *number* "/" TERM



A more practical approach

- Instead of

  E ::= *number* | E "+" E | E "–" E | E "*" E | E "/" E

- Write:

  E ::= *number* [Op *number*]*

  Op := "+" | "–" | "*" | "/"

- And the parser as:

```
parseNum(s);
while (!s.hasNext(opPat)) {
  s.next();
  parseNum(s);
}
```

A more practical approach

- What about operator precedence: * before +, etc?
- Grammar:

E ::= T [("+"|"-") T]*	Expression
T ::= F [("*"|"/") F]*	Term
F ::= number | "(" E ")"	Factor

- Parser:

```
public parseE(s) {
  parseT;
  while (!s.hasNext(addOpPat)) { // + or -
    s.next();
    parseT(s);
  } }
```

A more practical approach

- Now extend to build a parse tree
- `public Node parseE(Scanner s) {`

```
    Node t = parseT(Scanner s);
    while (!s.hasNext(addOpPat)) { // + or -
      String op = s.next();
      Node r = parseT(s);
      if ( op == “add” )
           t = new AddNode(t, r);
      else t = new SubNode(t, r);
     }
     return t;
  }
```

- What does the parse tree look like now?

Another look at empty strings

- How do we decide when to take an empty string branch?
- Suppose we want to parse a sequence of digits, each followed by a semicolon: “0;”, “2;1;1;5;”, etc.
- We can write the grammar like this:

  S ::= *digit* [“;” *digit*]* “;”

  or S ::= *digit* T “;”

  T ::= “;” *digit* T | “”
- If we read a digit, then a “;”, which branch do we take for T?
- Depends on what comes after the “;”, and we can’t see that!

Another look at empty strings

- This gives another condition for LL(1) grammars.

- LL(1) condition 2: If a nonterminal N can produce an empty string, then no token that can start an instance of N can also follow an instance of N.

- If we use […] and […]*, the condition extends to those as well.

- A grammar is called an LL(1) grammar if it satisfies LL(1) conditions 1 and 2.

Another look at empty strings

- We can rewrite the above grammar in LL(1) form:

S ::= *digit* “;” [*digit* “;”]*

or

S ::= *digit* “;” T
T ::= S | “”

- It’s not always that easy!

Next week: String searching

- How can you find an occurrence (all occurrences) of a string s in a text t?

- What is the cost?

- Can you make it faster??