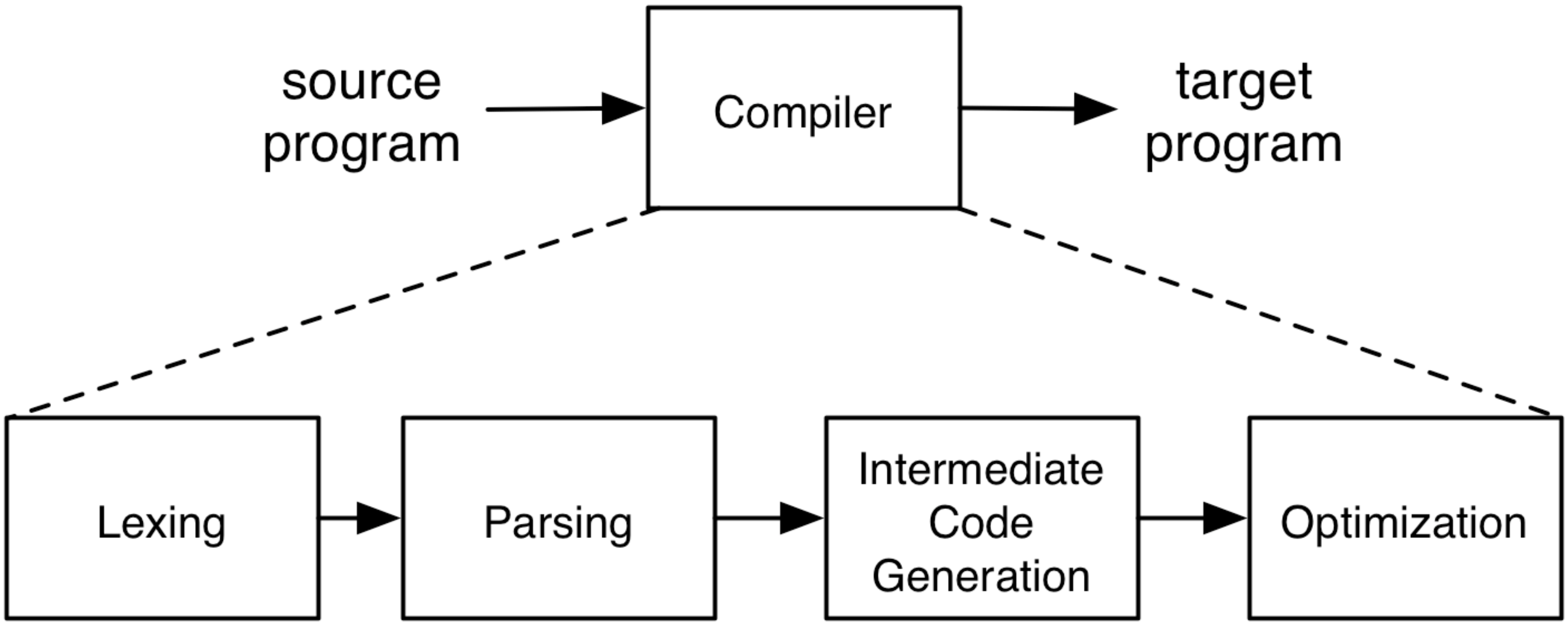
CMSC 330: Organization of Programming Languages

Parsing

Recall: Steps of Compilation



Implementing Parsers

- Many efficient techniques for parsing
  - I.e., turning strings into parse trees
  - Examples
    - LL(k), SLR(k), LR(k), LALR(k)…
    - Take CMSC 430 for more details
- One simple technique: recursive descent parsing
  - This is a top-down parsing algorithm
  - Other algorithms are bottom-up

Top-Down Parsing

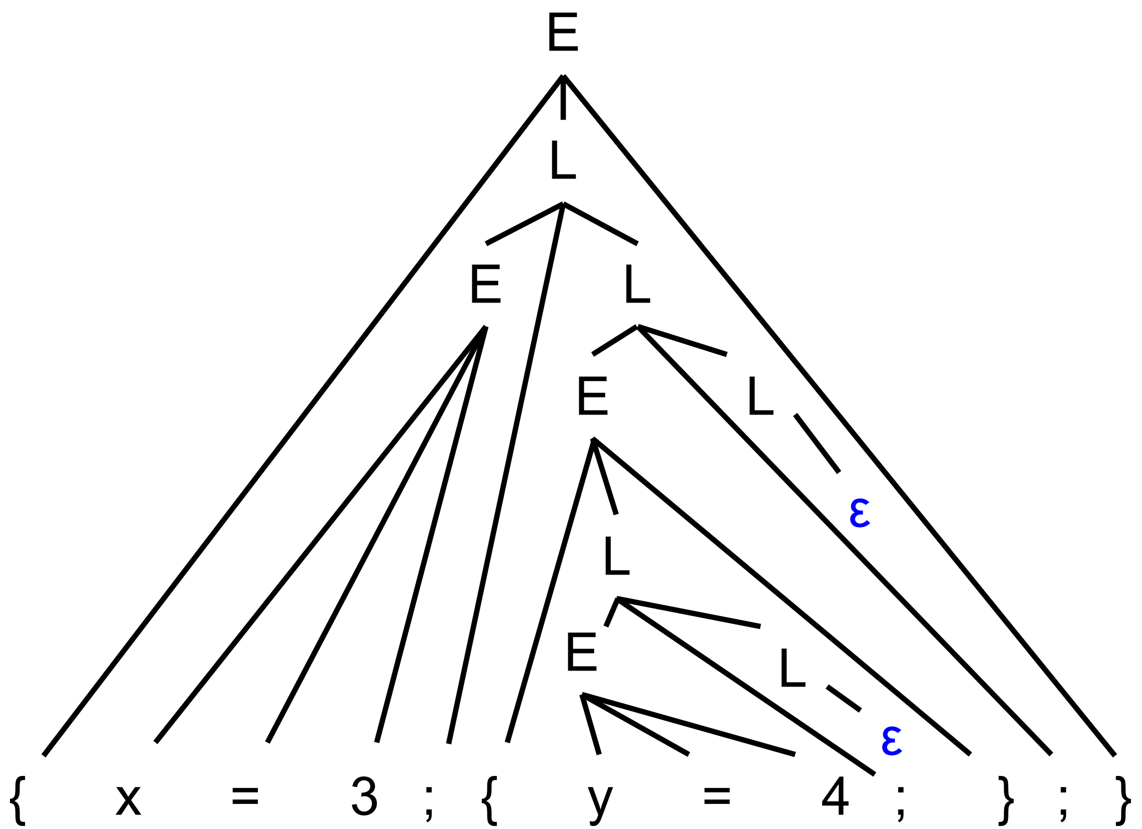
$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \varepsilon$

(Assume: id is variable name, n is integer)

Show parse tree for

{ x = 3 ; { y = 4 ; } ; }

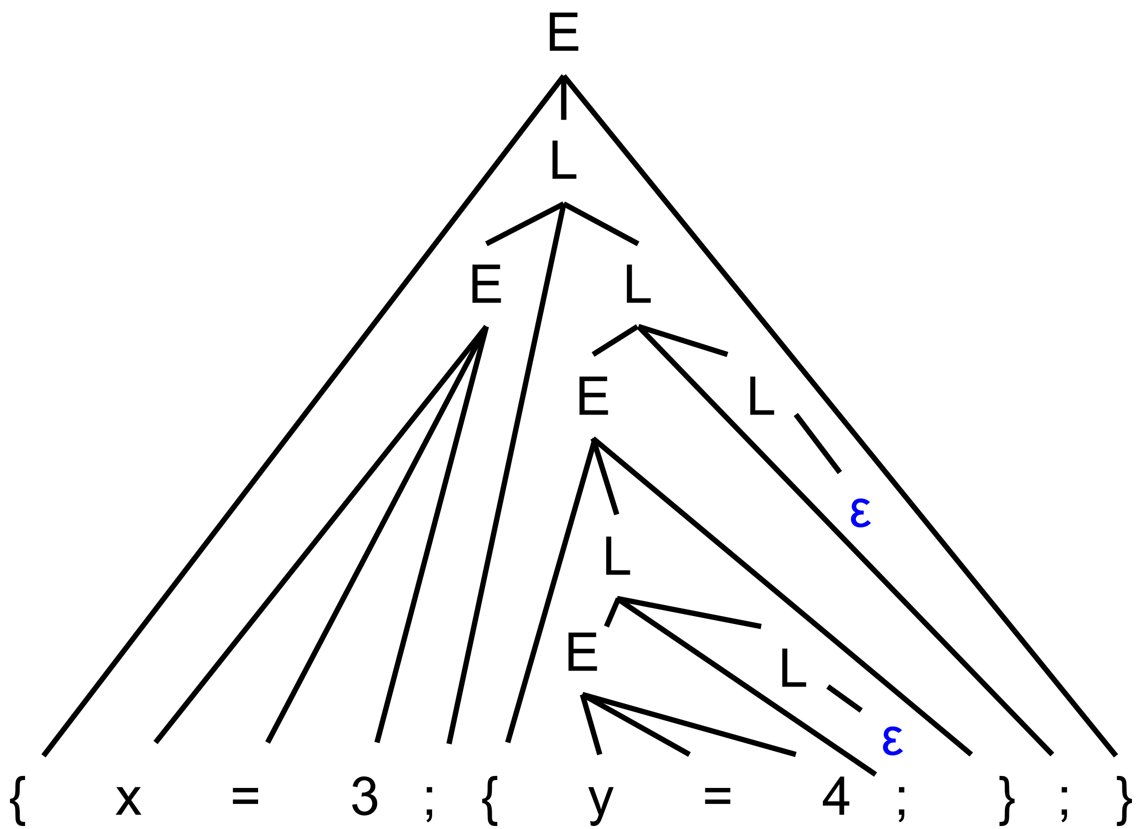


Bottom-up Parsing

$E \rightarrow$ id = n | { L }
$L \rightarrow$ E ; L | ε

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }

Note that final trees constructed are same as for top-down; only order in which nodes are added to tree is different



Example: Shift-Reduce Parsing

- Replaces RHS of production with LHS (nonterminal)
- Example grammar
  - $S \to aA$, $A \to Bc$, $B \to b$
- Example parse
  - $abc \Rightarrow aBc \Rightarrow aA \Rightarrow S$
  - Derivation happens in reverse
- Something to look forward to in CMSC 430
- Complicated to use; requires tool support
  - Bison, yacc produce shift-reduce parsers from CFGs

Tradeoffs

- Recursive descent parsers
  - Easy to write
    - The formal definition is a little clunky, but if you follow the code then it's almost what you might have done if you weren't told about grammars formally
  - Fast
    - Can be implemented with a simple table
- Shift-reduce parsers handle more grammars
  - Error messages may be confusing
- Most languages use hacked parsers (!)
  - Strange combination of the two

Recursive Descent Parsing

- Goal
  - Determine if we can produce the string to be parsed from the grammar's start symbol
- Approach
  - Recursively replace nonterminal with RHS of production
- At each step, we'll keep track of two facts
  - What tree node are we trying to match?
  - What is the lookahead (next token of the input string)?
    - Helps guide selection of production used to replace nonterminal

Recursive Descent Parsing (cont.)

- At each step, 3 possible cases
  - If we're trying to match a terminal
    - If the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we're trying to match a nonterminal
    - Pick which production to apply based on the lookahead
  - Otherwise fail with a parsing error

Parsing Example

$E \rightarrow$ id = n | { L }

$L \rightarrow$ E ; L | ε

- Here n is an integer and id is an identifier

- One input might be
  - { x = 3; { y = 4; }; }
  - This would get turned into a list of tokens

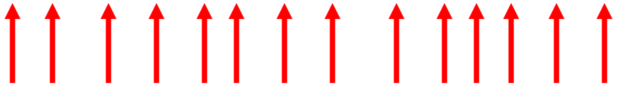
    { x = 3 ; { y = 4 ; } ; }
  - And we want to turn it into a parse tree

Parsing Example (cont.)

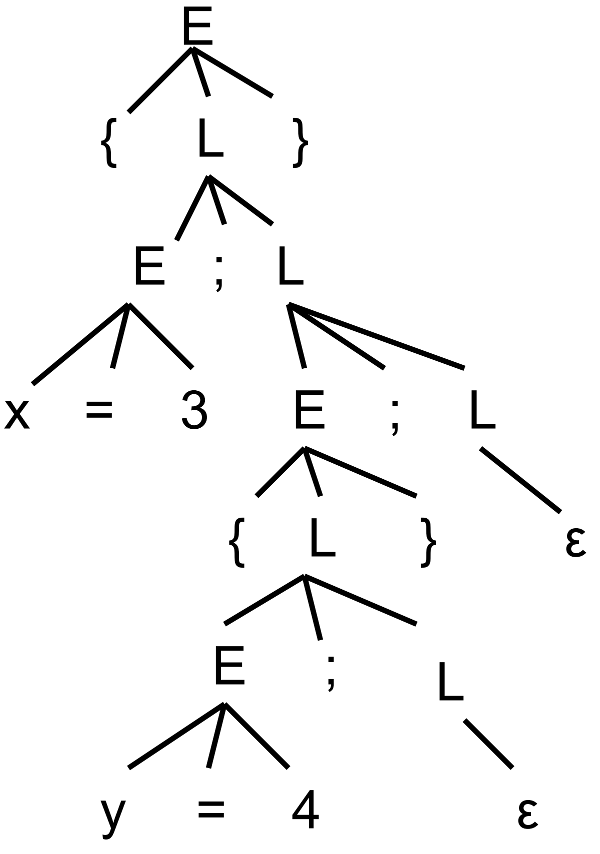
$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E \,;\, L \mid \varepsilon$

{ x = 3 ; { y = 4 ; } ; }



lookahead



Recursive Descent Parsing (cont.)

- Key step
  - Choosing which production should be selected
- Two approaches
  - Backtracking
    - Choose some production
    - If fails, try different production
    - Parse fails if all choices fail
  - Predictive parsing
    - Analyze grammar to find FIRST sets for productions
    - Compare with lookahead to decide which production to select
    - Parse fails if lookahead does not match FIRST

First Sets

- Motivating example
  - The lookahead is x
  - Given grammar S → xyz | abc
    - Select S → xyz since 1st terminal in RHS matches x
  - Given grammar S → A | B A → x | y B → z
    - Select S → A, since A can derive string beginning with x
- In general
  - Choose a production that can derive a sentential form beginning with the lookahead
  - Need to know what terminal may be first in any sentential form derived from a nonterminal / production

First Sets

- Definition
  - First(γ), for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to
  - We'll use this to decide what production to apply
- Examples
  - Given grammar S → xyz | abc
    - First(xyz) = { x }, First(abc) = { a }
    - First(S) = First(xyz) U First(abc) = { x, a }
  - Given grammar S → A | B   A → x | y   B → z
    - First(x) = { x }, First(y) = { y }, First(A) = { x, y }
    - First(z) = { z }, First(B) = { z }
    - First(S) = { x, y, z }

Calculating First(γ)

- For a terminal a
  - First(a) = { a }
- For a nonterminal N
  - If $N \rightarrow \varepsilon$, then add ε to First(N)
  - If $N \rightarrow \alpha_1\ \alpha_2\ \ldots\ \alpha_n$, then (note the α_i are all the symbols on the right side of one single production):
    - Add First($\alpha_1\alpha_2 \ldots \alpha_n$) to First(N), where First($\alpha_1\ \alpha_2 \ldots \alpha_n$) is defined as
      - First(α_1) if $\varepsilon \notin$ First(α_1)
      - Otherwise (First(α_1) – ε) $\cup$ First($\alpha_2 \ldots \alpha_n$)
    - If $\varepsilon \in$ First(α_i) for all i, $1 \leq i \leq k$, then add ε to First(N)

First( ) Examples

$E \rightarrow$ id = n | { L }
$L \rightarrow$ E ; L | ε

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { id, "{" }
First(L) = { id, "{", ε }

$E \rightarrow$ id = n | { L } | ε
$L \rightarrow$ E ; L

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { id, "{", ε }
First(L) = { id, "{", ";" }

Recursive Descent Parser Implementation

- For terminals, create function match(a)
  - If lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Otherwise fails with a parse error if lookahead is not a
  - In algorithm descriptions, consider parse_a, parse_term(a) to be aliases for match(a)
- For each nonterminal N, create a function parse_N
  - Called when we're trying to parse a part of the input which corresponds to (or can be derived from) N
  - parse_S for the start symbol S begins the parse

Parser Implementation (cont.)

- The body of parse_N for a nonterminal N does the following
  - Let $N \rightarrow \beta_1 \mid \ldots \mid \beta_k$ be the productions of N
    - Here β_i is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production $N \rightarrow \beta_i$ such that the lookahead is in First(β_i)
    - It must be that $\mathrm{First}(\beta_i) \cap \mathrm{First}(\beta_j) = \varnothing$ for $i \neq j$
    - If there is no such production, but $N \rightarrow \varepsilon$ then return
    - Otherwise fail with a parse error
  - Suppose $\beta_i = \alpha_1\ \alpha_2 \ldots \alpha_n$. Then call parse_$\alpha_1$(); ... ; parse_$\alpha_n$() to match the expected right-hand side, and return

Parser Implementation (cont.)

- Parse is built on procedure calls
- Procedures may be (mutually) recursive

Recursive Descent Parser

- Given grammar $S \rightarrow xyz \mid abc$
  - First(xyz) = { x }, First(abc) = { a }
- Parser

```
parse_S( ) {
    if (lookahead == "x") {
        match("x"); match("y"); match("z");    // S → xyz
    }
    else if (lookahead == "a") {
        match("a"); match("b"); match("c");    // S → abc
    }
    else error( );
}
```

Recursive Descent Parser

- Given grammar $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$
  - First(A) = { x, y }, First(B) = { z }
- Parser

```
parse_S( ) {
    if ((lookahead == “x”) ||
      (lookahead == “y”))
      parse_A( );     // S → A
    else if (lookahead == “z”)
      parse_B( );     // S → B
    else error( );
}
```

```
parse_A( ) {
    if (lookahead == “x”)
      match(“x”);     // A → x
    else if (lookahead == “y”)
      match(“y”);     // A → y
    else error( );
}
parse_B( ) {
    if (lookahead == “z”)
      match(“z”);     // B → z
    else error( );
}
```

Example

$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \varepsilon$

First(E) = { id, "{" }

```
parse_E( ) {
    if (lookahead == "id") {
      match("id");
      match("=");    // E → id = n
      match("n");
    }
    else if (lookahead == "{") {
      match("{");
      parse_L( );    // E → { L }
      match("}");
    }
    else error( );
}
```

```
parse_L( ) {
    if ((lookahead == "id") ||
      (lookahead == "{")) {
      parse_E( );
      match(";"); // L → E ; L
      parse_L( );
    }
    else ;              // L → ε
}
```

Things to Notice

- If you draw the execution trace of the parser
  - You get the parse tree
- Examples
  - Grammar

    S → xyz

    S → abc
  - String "xyz"

    parse_S( )
      match("x")
      match("y")
      match("z")

```
  S
 /|\
x y z
```

  - Grammar

    S → A | B

    A → x | y

    B → z
  - String "x"

    parse_S( )
      parse_A( )
        match("x")

```
S
|
A
|
x
```

Things to Notice (cont.)

- This is a predictive parser
  - Because the lookahead determines exactly which production to use
- This parsing strategy may fail on some grammars
  - Production First sets overlap
  - Production First sets contain ε
  - Possible infinite recursion
- Does not mean grammar is not usable
  - Just means this parsing method not powerful enough
  - May be able to change grammar

Conflicting FIRST Sets

- Consider parsing the grammar $E \rightarrow ab \mid ac$
  - First(ab) = a
  - First(ac) = a

  Parser cannot choose between RHS based on lookahead!
- Parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and
  - $First(\alpha_1) \cap First(\alpha_2)$!= ε or $\varnothing$
- Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \ldots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Repeat as necessary
- Examples
  - $S \rightarrow ab \mid ac$ ⇨ $S \rightarrow aL$ $\quad L \rightarrow b \mid c$
  - $S \rightarrow abcA \mid abB \mid a$ ⇨ $S \rightarrow aL$ $\quad L \rightarrow bcA \mid bB \mid \varepsilon$
  - $L \rightarrow bcA \mid bB \mid \varepsilon$ ⇨ $L \rightarrow bL' \mid \varepsilon$ $\quad L' \rightarrow cA \mid B$

Alternative Approach

- Change structure of parser
  - First match common prefix of productions
  - Then use lookahead to chose between productions
- Example
  - Consider parsing the grammar $E \rightarrow a+b \mid a*b \mid a$

```
parse_E( ) {
    match("a");                    // common prefix
    if (lookahead == "+") {        // E → a+b
      match("+"); match("b");      }
    if (lookahead == "*") {        // E → a*b
      match("*"); match("b");      }
    else { }                       // E → a
}
```

Left Recursion

- Consider grammar $S \rightarrow Sa \mid \varepsilon$
  - Try writing parser

```
parse_S( ) {
    if (lookahead == "a") {
        parse_S( );
        match("a");   // S → Sa
    }
    else { }
}
```

  - Body of parse_S( ) has an infinite loop
    - if (lookahead = "a") then parse_S( )
  - Infinite loop occurs in grammar with left recursion

Right Recursion

- Consider grammar $S \rightarrow aS \mid \varepsilon$
  - Again, First(aS) = a
  - Try writing parser

```
parse_S( ) {
    if (lookahead == "a") {
      match("a");
      parse_S( );    // S → aS
    }
    else { }
}
```

  - Will parse_S( ) infinite loop?
    - Invoking match( ) will advance lookahead, eventually stop
  - Top down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
    - β must exist or derivation will not yield string
- Rewrite grammar as (repeat as needed)
  - $A \rightarrow \beta L$
  - $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon$
- Replaces left recursion with right recursion
- Examples
  - $S \rightarrow Sa \mid \varepsilon$ ⇨ $S \rightarrow L$ $L \rightarrow aL \mid \varepsilon$
  - $S \rightarrow Sa \mid Sb \mid c$ ⇨ $S \rightarrow cL$ $L \rightarrow aL \mid bL \mid \varepsilon$

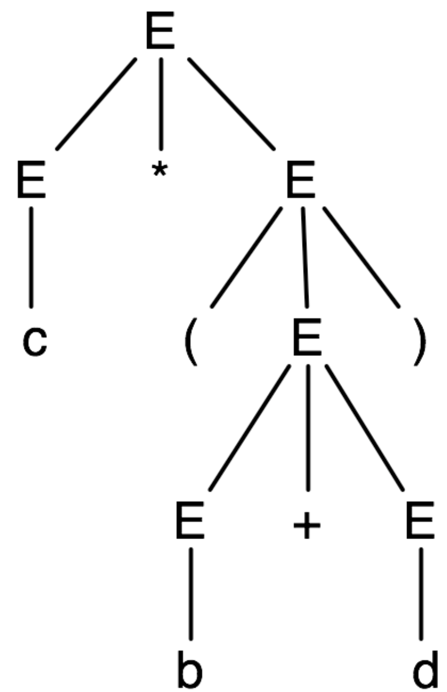
What’s Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    - Parentheses
    - Extra nonterminals for precedence
  - This extra stuff is needed for parsing

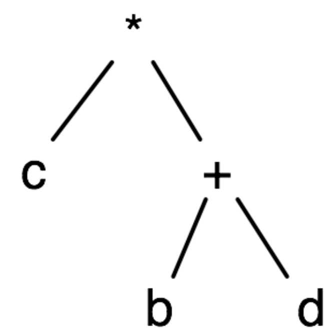
- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts



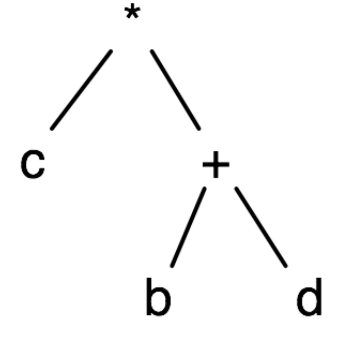
parse tree



AST

Abstract Syntax Trees (cont.)

- Intuitively, ASTs correspond to the data structure you'd use to represent strings in the language
  - Note that grammars describe trees
  - So do OCaml datatypes (which we'll see later)
  - E → a | b | c | E+E | E-E | E*E | (E)



Producing an AST

- To produce an AST, we can modify the parse() functions to construct the AST along the way
  - match(a) returns an AST node (leaf) for a
  - Parse_A returns an AST node for A
    - AST nodes for RHS of production become children of LHS node
- Example
  - $S \rightarrow aA$

```
Node parse_S( ) {
      Node n1, n2;
      if (lookahead == "a") {
         n1 = match("a");
         n2 = parse_A( );
         return new Node(n1,n2);
      }
}
```

```
   S
  / \
 a   A
     |
```

The Compilation Process