# CMSC 330: Organization of Programming Languages

Parsing, con't.



## 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
    - $S \rightarrow xyz$
    - $S \rightarrow abc$
  - String “xyz”

```
parse_S()
  match(“x”)
  match(“y”)
  match(“z”)
```

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

  - Grammar
    - $S \rightarrow A \mid B$
    - $A \rightarrow x \mid y$
    - $B \rightarrow z$
  - 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
  - Possible infinite recursion
  - Production First sets overlap
  - Production First sets contain ε
- This does not mean the grammar is not usable- it just means this parsing method is not powerful enough
  - You may be able to change the grammar

## Left Factoring

- Consider parsing the grammar $E \rightarrow ab \mid ac$
  - First(ab) = a
  - First(ac) = a
  - The parser cannot choose between right-hand sides based on the lookahead!
- A recursive descent parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and $\text{First}(\alpha_1) \cap \text{First}(\alpha_2)$ != $\varepsilon$ or $\varnothing$
- Solution: rewrite the 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 it as:

  $A \rightarrow xL \mid \beta$

  $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Repeat as necessary
- Examples:

| | | |
|---|---|---|
| $S \rightarrow ab \mid ac$ | $\Rightarrow S \rightarrow aL$ | $L \rightarrow b \mid c$ |
| $S \rightarrow abcA \mid abB \mid a$ | $\Rightarrow S \rightarrow aL$ | $L \rightarrow bcA \mid bB \mid \varepsilon$ |
| $L \rightarrow bcA \mid bB \mid \varepsilon$ | $\Rightarrow L \rightarrow bL' \mid \varepsilon$ | $L' \rightarrow cA \mid B$ |



## Left Recursion

- Consider grammar $S \rightarrow Sa \mid \varepsilon$
  - First(Sa) = a, so we're ok as far as which production
  - 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 the lookahead, eventually stop
  - Top down parsers handles grammars with *right recursion*

## Algorithm To Eliminate Left Recursion

- Given grammar $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
  - (Why must $\beta$ exist?)
- Rewrite the grammar as:

  $A \rightarrow \beta L$

  $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon$
- Replaces left recursion with right recursion
- Repeat as necessary



## Eliminating Left Recursion (cont.)

- Examples

  $S \rightarrow Sa \mid \varepsilon \quad \Rightarrow S \rightarrow L \quad L \rightarrow aL \mid \varepsilon$

  $S \rightarrow Sa \mid Sb \mid c \quad \Rightarrow S \rightarrow cL \quad L \rightarrow aL \mid bL \mid \varepsilon$

- May need more powerful algorithms to eliminate *mutual recursion* leading to left recursion

  $S \rightarrow Aa \mid b$

  $A \rightarrow Sb$



## Expression Grammar for Top-Down Parsing

$E \rightarrow T\ E'$

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

$T \rightarrow P\ \ T'$

$T' \rightarrow \varepsilon \mid *\ T$

$P \rightarrow n \ \mid\ (E)$

- Notice we can always decide what production to choose with only one symbol of lookahead



## Tradeoffs with Other Approaches

- Recursive descent parsers are 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
  - They're unable to handle certain kinds of grammars
- Recursive descent is good for a simple parser
  - Though tools can be fast if you’re familiar with them
- Can implement top-down predictive parsing as a table-driven parser, by maintaining an explicit stack to track progress

# Tradeoffs with Other Approaches

- More powerful techniques need tool support
  - Can take time to learn tools
- The main alternative is a bottom-up, shift-reduce parser
  - Replaces RHS of production with LHS (nonterminal)
  - Example grammar: S → aA, A → Bc, B → b
  - Example parse: abc ⇒ aBc ⇒ aA ⇒ S
    - The derivation happens in reverse
  - Something to look forward to in CMSC 430



# Usage of Parse Trees

- Parse trees contain too much information
  - E.g., they have parentheses and they have extra nonterminals for precedence
  - This extra stuff is needed for parsing

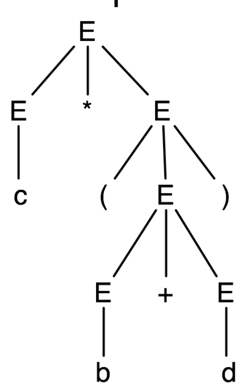
- But when we want to reason about languages, it gets in the way (it's too much detail)

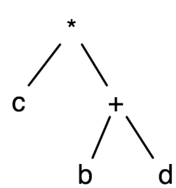


# Abstract Syntax Trees (ASTs)

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

parse tree



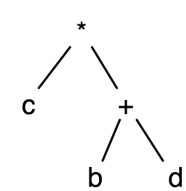

AST



# ASTs (cont'd)

- Intuitively, ASTs correspond to the data structure you'd use to represent strings in a language
  - Note that grammars describe trees, and so do OCaml datatypes
  - This example uses the first expression grammar, E → a | b | c | E+E | E-E | E*E | (E), for simplicity

```
type ast =
  Letter of char
| Plus of ast * ast
| Minus of ast * ast
| Times of ast * ast
```



# 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