CSE443 Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

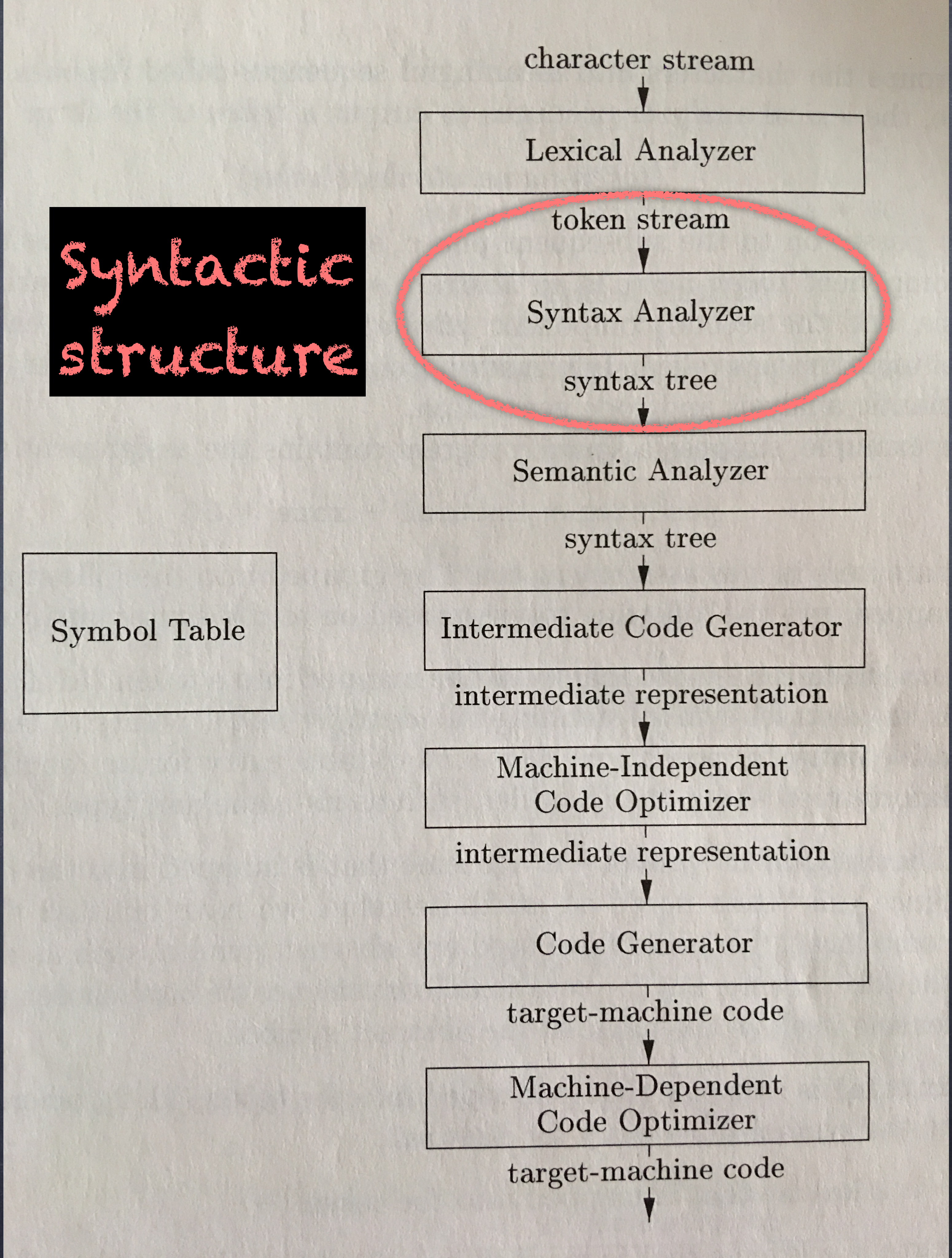


Figure 1.6, page 5 of text

Recursion and parentheses

- To generate 2+3*4 or 3*4+2, the parse tree is built so that + is higher in the tree than *.
- To force an addition to be done prior to a multiplication we must use parentheses, as in (2+3)*4.
- Grammar captures this in the recursive case of an expression, as in the following grammar fragment:

  <expr> → <expr> + <term> | <term>

  <term> → <term> * <factor> | <factor>

  <factor> → <variable> | <constant> | "(" <expr> ")"

RL ⊆ CFL

- Given a regular language L we can always construct a context free grammar G such that L = $\mathcal{L}$(G).
- For every regular language L there is an NFA M = (S,Σ,δ,F,s_0) such that L = $\mathcal{L}$(M).
- Build G = (N,T,P,S_0) as follows:
  - $N = \{ N_s \mid s \in S \}$
  - $T = \{ t \mid t \in \Sigma \}$
  - If $\delta(i,a)=j$, then add $N_i \to a\, N_j$ to P
  - If $i \in F$, then add $N_i \to \varepsilon$ to P
  - $S_0 = N_{s_0}$

RL $\subsetneq$ CFL

- Show that not all CF languages are regular.
- To do this we only need to demonstrate that there exists a CFL that is not regular.
- Consider $L = \{ a^n b^n \mid n \geq 1 \}$
- Claim: $L \in CFL$, $L \notin RL$

Relevance?
Nested '{' and '}'

```
public class Foo {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++) {
        if (args[I].length() < 3) { … }
       else { … }
    }
  }
}
```

Context Free Grammars and parsing

- $O(n^3)$ algorithms to parse any CFG exist
- Programming language constructs can generally be parsed in $O(n)$

Top-down & bottom-up

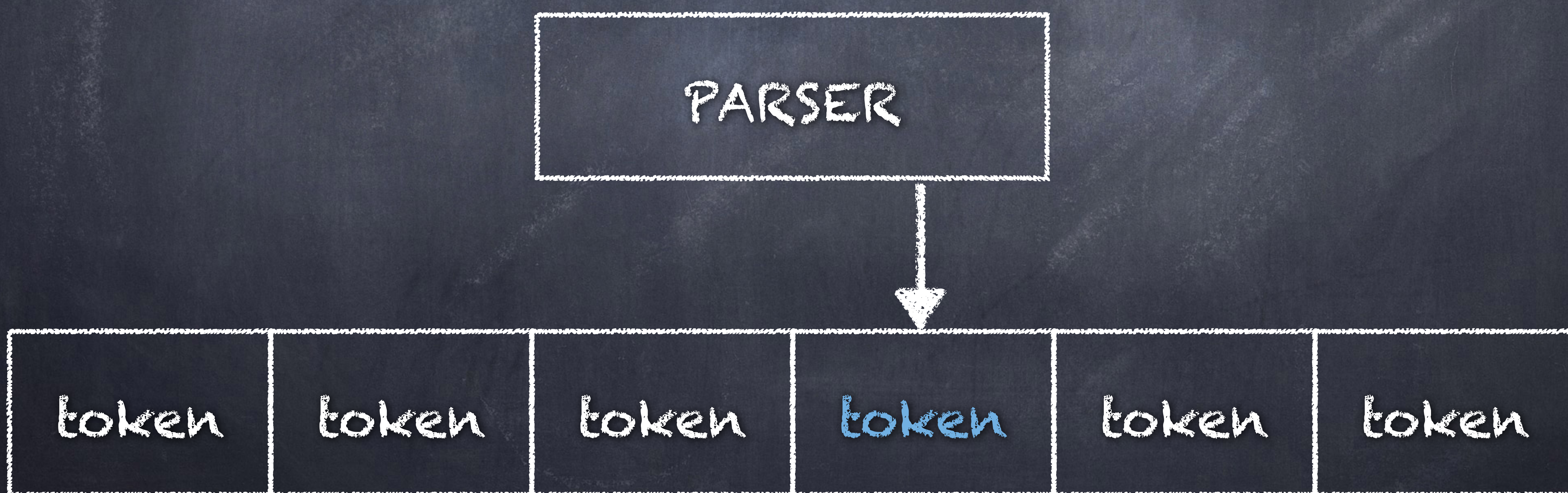
- A top-down parser builds a parse tree from root to the leaves
  - easier to construct by hand
- A bottom-up parser builds a parse tree from leaves to root
  - Handles a larger class of grammars
  - tools (yacc/bison) build bottom-up parsers

Our presentation
First top-down, then bottom-up

- Present top-down parsing first.
- Introduce necessary vocabulary and data structures.
- Move on to bottom-up parsing second.

vocab: look-ahead

- The current symbol being scanned in the input is called the lookahead symbol.



Top-down parsing

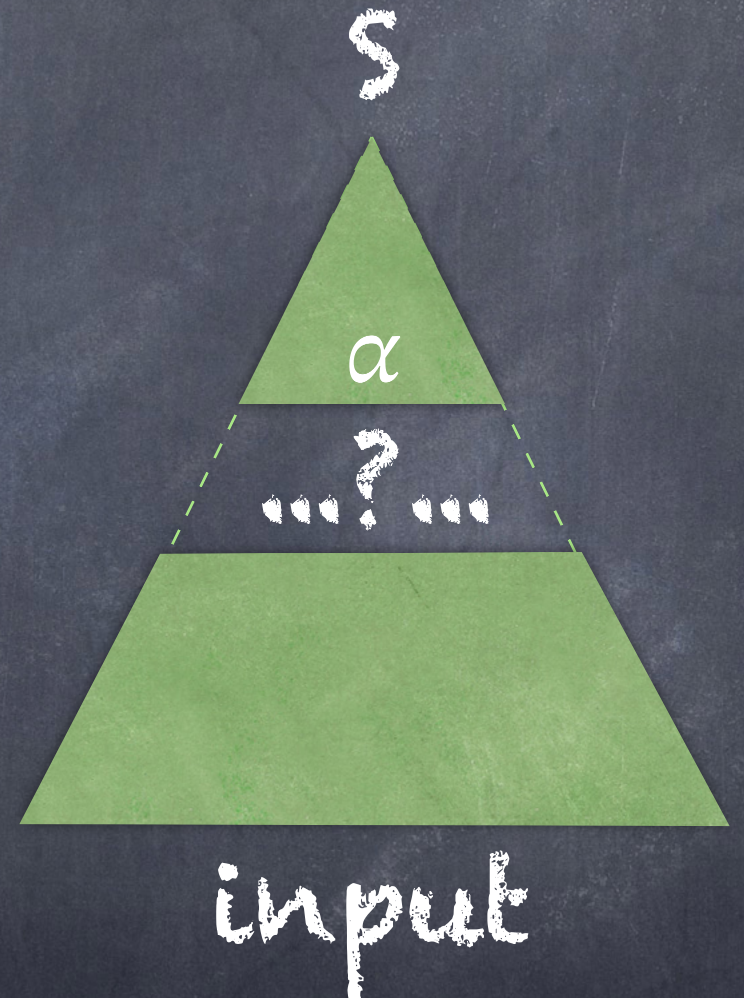
- Start from grammar's start symbol
- Build parse tree so its yield matches input
- predictive parsing: a simple form of recursive descent parsing

Basic idea: try to build a derivation S =>* input

S =>* α

...?...

=>* input



FIRST(α)

- If $\alpha \in (N \cup T)^*$ then FIRST(α) is "the set of terminals that appear as the first symbols of one or more strings of terminals generated from α." [p. 64]
- Ex: If A -> a β then FIRST(A) = {a}
- Ex. If A -> a β | B then FIRST(A) = {a} $\cup$ FIRST(B)

FIRST(α)

- First sets are considered when there are two (or more) productions to expand $A \in N$: $A \rightarrow \alpha \mid \beta$
- Predictive parsing requires that $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$

ε productions

- If lookahead symbol does not match first set, use ε production not to advance lookahead symbol but instead "discard" non-terminal:
  - optexpt -> expr | ε
- "While parsing optexpr, if the lookahead symbol is not in FIRST(expr), then the ε production is used" [p. 66]

Left recursion

- Grammars with left recursion are problematic for top-down parsers, as they lead to infinite regress.

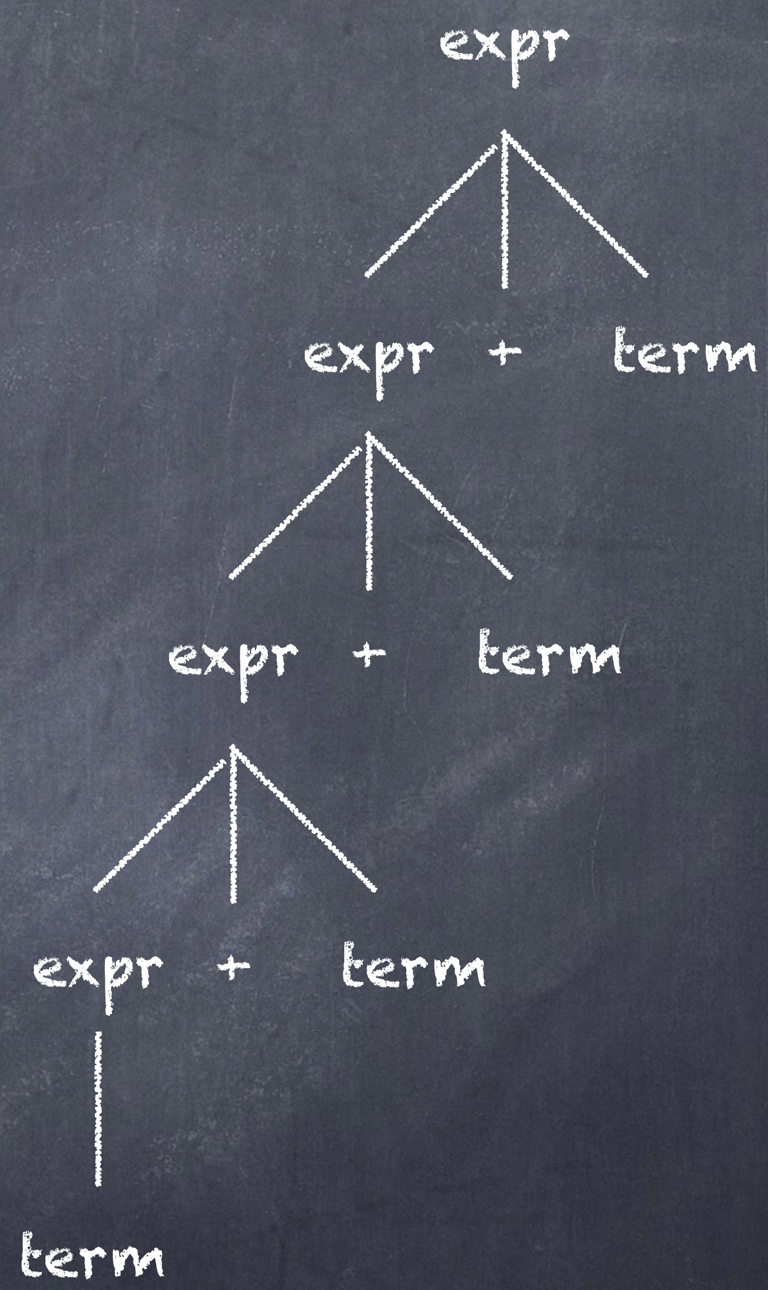
Left recursion example

- Grammar:

    expr -> expr + term | term

    term -> id

- FIRST sets for rule alternatives are not disjoint:
    - FIRST(expr) = id
    - FIRST(term) = id



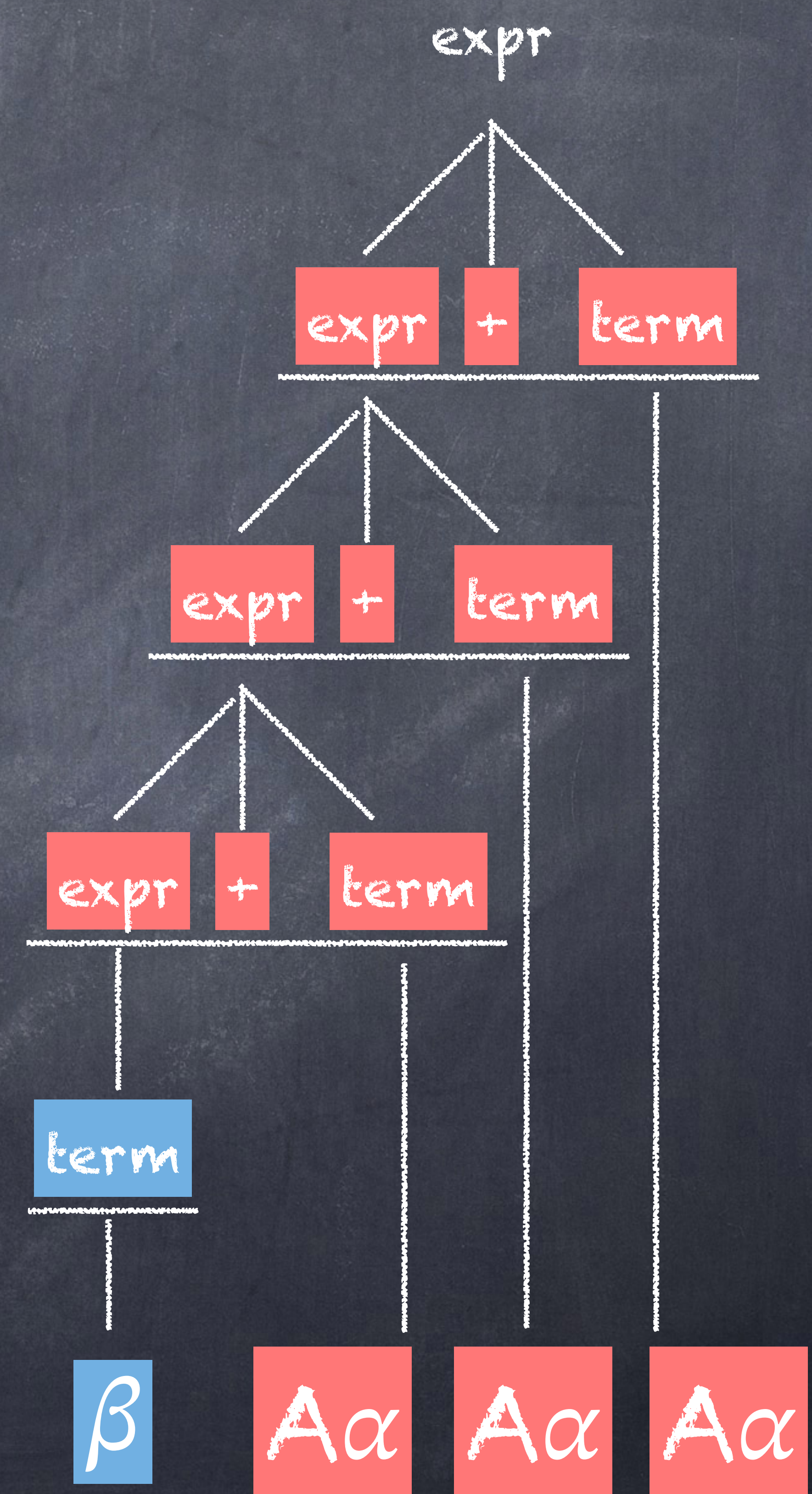
Left recursion example

- Grammar: A α β

  expr -> expr + term | term

  term -> id

- FIRST sets for rule alternatives are not disjoint:
  - FIRST(expr) = id
  - FIRST(term) = id

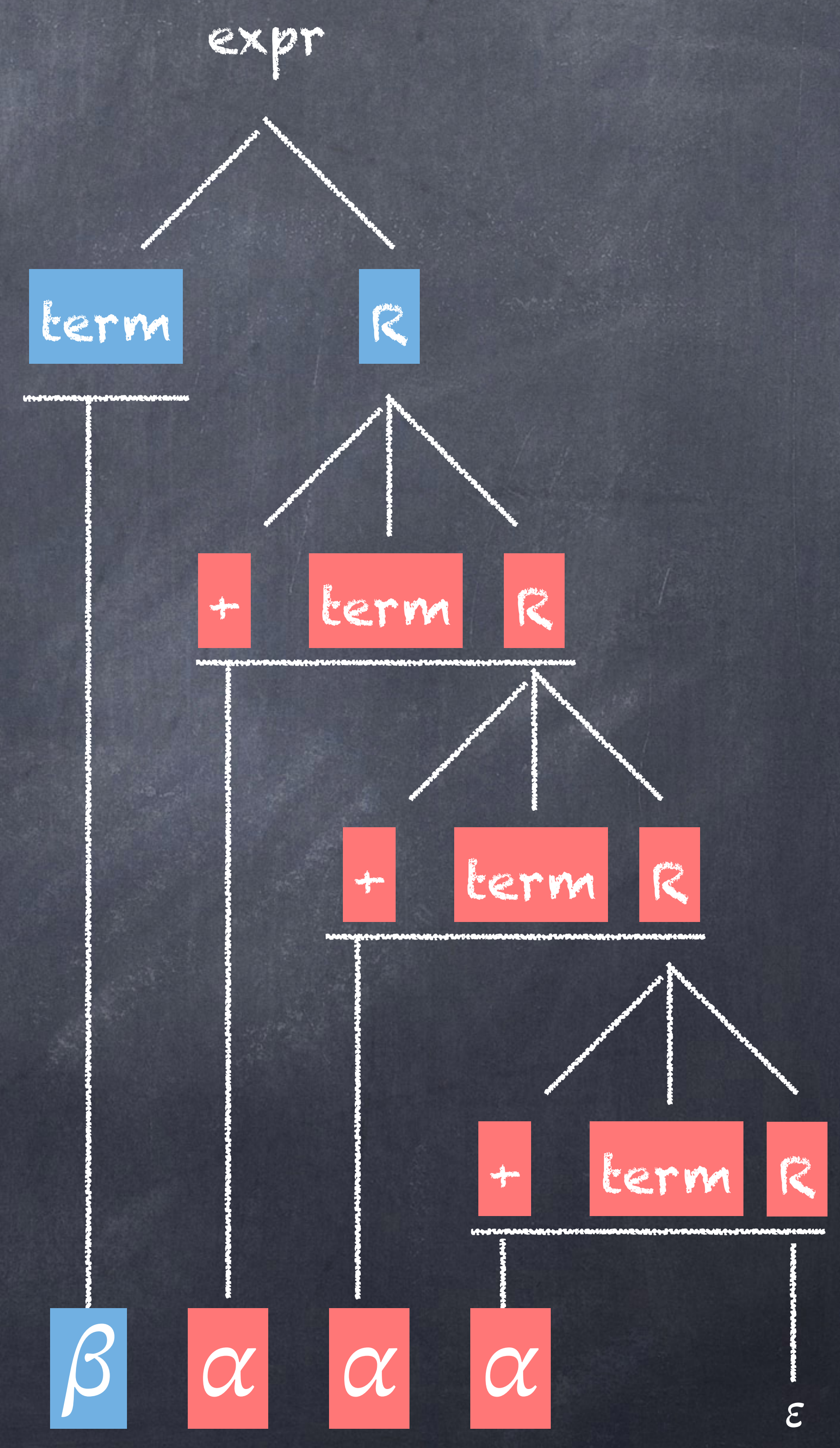


Rewriting grammar to remove left recursion

- expr rule is of form A -> A α | β
- Rewrite as two rules
  - A -> β R
  - R -> α R | ε

Back to example

- Grammar is re-written as
  - expr -> term R
  - R -> + term R | ε



Ambiguity

- A grammar G is ambiguous if $\exists\ \sigma \in \mathcal{L}(G)$ that has two or more distinct parse trees.
- Example - dangling 'else':

if <expr> then if <expr> then <stmt> else <stmt>

if <expr> then { if <expr> then <stmt> } else <stmt>

if <expr> then { if <expr> then <stmt> else <stmt> }

dangling else resolution

- usually resolved so else matches closest if-then
- we can re-write grammar to force this interpretation (ms = matched statement, os = open statement)
  - <stmt> -> <ms> | <os>
  - <ms> -> if <expr> then <ms> else <ms> | ...
  - <os> -> if <expr> then <stmt> | if <expr> then <ms> else <os>

Left factoring

- If two (or more) rules share a prefix then their FIRST sets do not distinguish between rule alternatives.
- If there is a choice point later in the rule, rewrite rule by factoring common prefix
- Example: rewrite

$$A \to \alpha\ \beta_1 \mid \alpha\ \beta_2$$

- as

$$A \to \alpha\ A'$$

$$A' \to \beta_1 \mid \beta_2$$

Predictive parsing:

a special case of recursive-descent parsing that does not require backtracking

Each non-terminal A ∈ N has an associated procedure:

```
void A() {
    choose an A-production A -> X1 X2 ... Xk
    for (i = 1 to k) {
        if (xi ∈ N) {
            call xi()
        }
        else if (xi = current input symbol) {
            advance input to next symbol
        }
        else error
    }
}
```

Predictive parsing:

a special case of recursive-descent parsing that does not require backtracking

Each non-terminal A ∈ N has an associated procedure:

```
void A() {
    choose an A-production A -> X1 X2 ... Xk
    for (i = 1 to k) {
        if (xi ∈ N) {
            call xi()
        }
        else if (xi = current input symbol) {
            advance input to next symbol
        }
        else error
    }
}
```

There is non-determinism in choice of production. If "wrong" choice is made the parser will need to revisit its choice by backtracking.

A predictive parser can always make the correct choice here.

FIRST(X)

- if $X \in T$ then FIRST(X) = { X }
- if $X \in N$ and $X \rightarrow Y_1\ Y_2 \dots Y_k \in P$ for $k \geq 1$, then
  - add $a \in T$ to FIRST(X) if $\exists i$ s.t. $a \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) $\forall\ j < i$ (i.e. $Y_1\ Y_2 \dots Y_{i-1} \Rightarrow^* \varepsilon$)
  - if $\varepsilon \in$ FIRST(Y_j) $\forall\ j <= k$ add ε to FIRST(X)
  - if $X \rightarrow \varepsilon \in P$, then add ε to FIRST(X)

FOLLOW(X)

- Place \$ in FOLLOW(S), where S is the start symbol (\$ is an end marker)
- if A -> $\alpha B \beta \in P$, then FIRST(β) - $\{\varepsilon\}$ is in FOLLOW(B)
- if A -> $\alpha B \in P$ or A -> $\alpha B \beta \in P$ where $\varepsilon \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B)

Table-driven predictive parsing Algorithm 4.32 (p. 224)

- INPUT: Grammar G = (N,T,P,S)
- OUTPUT: Parsing table M
- For each production A -> α of G:

1. For each terminal a $\in$ FIRST(α), add A -> α to M[A,a]
2. If $\varepsilon \in$ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
3. If $\varepsilon \in$ FIRST(α) and \$ $\in$ FOLLOW(A), add A -> α to M[A,\$]