CSE443
Compilers

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

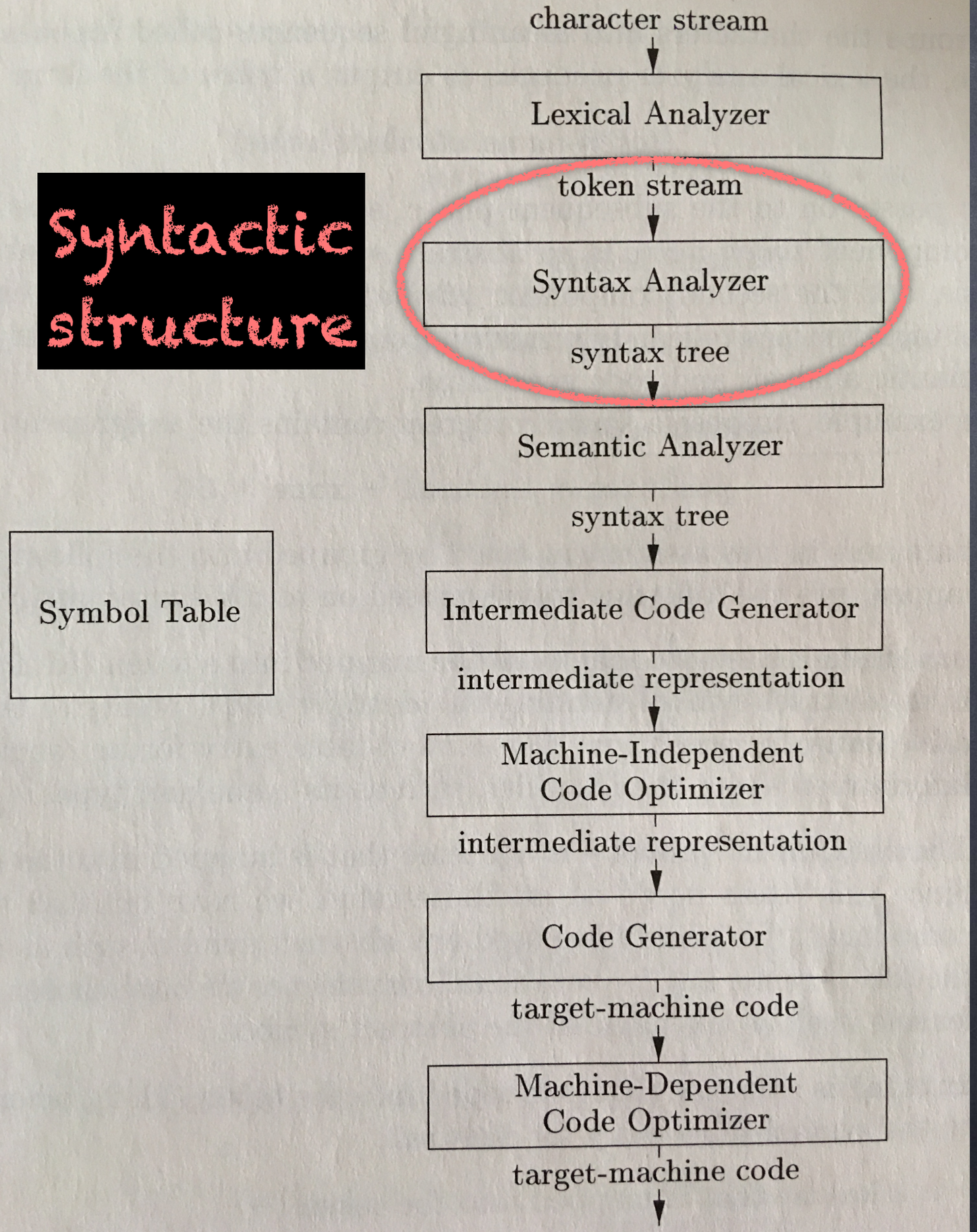
Project progress

- User stories
- Story -> task decomposition
- Meet with PMs tomorrow

Phases of a compiler

Figure 1.6, page 5 of text

Syntactic structure



Sample grammars

- http://www.schemers.org/Documents/Standards/R5RS/HTML/
- https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ref_002dsyn_002dsyn_002dsen.html
- https://docs.oracle.com/javase/specs/jls/se13/html/jls-19.html
- http://blackbox.userweb.mwn.de/Pascal-EBNF.html
- https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

Language Semantics

What's in language specification?

What's left up to the language implementor?

6.2.2 Evaluation Order

The order of evaluation of subexpressions within an expression is undefined. In particular, you cannot assume that the expression is evaluated left to right. For example:

```
int x = f(2) +g(3);     // undefined whether f() or g() is called first
```

C++ Programming Language, 3rd edition.
Bjarne Stroustrup. (c) 1997. Page 122.

A compiler translates high level language statements into a much larger number of low-level statements, and then applies optimizations. The entire translation process, including optimizations, must preserve the semantics of the original high-level program.

By not specifying the order in which subexpressions are evaluated (left-to-right or right-to-left) a C++ compiler can potentially re-order the resulting low-level instructions to give a "better" result.

The C++ programming language, 3rd edition, (c) 1997, Bjarne Stroustrup, pg 74

4.6 Sizes

Some of the aspects of C++'s fundamental types, such as the size of an *int*, are implementation-defined (§C.2). I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. People who claim they don't care about portability usually do so because they use only a single system and feel they can afford the attitude that ''the language is what my compiler implements.'' This is a narrow and shortsighted view. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favorite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies

The reason for providing more than one integer type, more than one unsigned type, and more than one floating-point type is to allow the programmer to take advantage of hardware characteristics. On many machines, there are significant differences in memory requirements, memory access times, and computation speed between the different varieties of fundamental types. If you know a machine, it is usually easy to choose, for example, the appropriate integer type for a particular variable. Writing truly portable low-level code is harder.

Sizes of C++ objects are expressed in terms of multiples of the size of a *char*, so by definition the size of a *char* is *1*. The size of an object or type can be obtained using the *sizeof* operator (§6.2). This is what is guaranteed about sizes of fundamental types:

$$1 \equiv sizeof(char) \le sizeof(short) \le sizeof(int) \le sizeof(long)$$

$$1 \le sizeof(bool) \le sizeof(long)$$

$$sizeof(char) \le sizeof(wchar\_t) \le sizeof(long)$$

$$sizeof(float) \le sizeof(double) \le sizeof(long\ double)$$

$$sizeof(N) \equiv sizeof(signed\ N) \equiv sizeof(unsigned\ N)$$

where *N* can be ***char***, ***short int***, ***int***, or ***long int***. In addition, it is guaranteed that a ***char*** has at least 8 bits, a ***short*** at least 16 bits, and a ***long*** at least 32 bits. A ***char*** can hold a character of the machine's character set.

The precision of these objects depends on the machine at hand; the table below shows some representative values.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 bits	9 bits	8 bits	8 bits
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64

The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will normally reflect the most "natural" size for a particular machine. As you can see, each compiler is free to interpret `short` and `long` as appropriate for its own hardware. About all you should count on is that `short` is no longer than `long`.

Review

Programming Language Grammar Fragment

```
<program> -> <stmt-list>
<stmt-list> -> <stmt> | <stmt> ; <stmt-list>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const
```

Notes:
<var> is defined in the grammar
const is not defined in the grammar

derivations of
a = b + const

grammar

```
<program> -> <stmt-list>
<stmt-list> -> <stmt> | <stmt> ; <stmt-list>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const
```

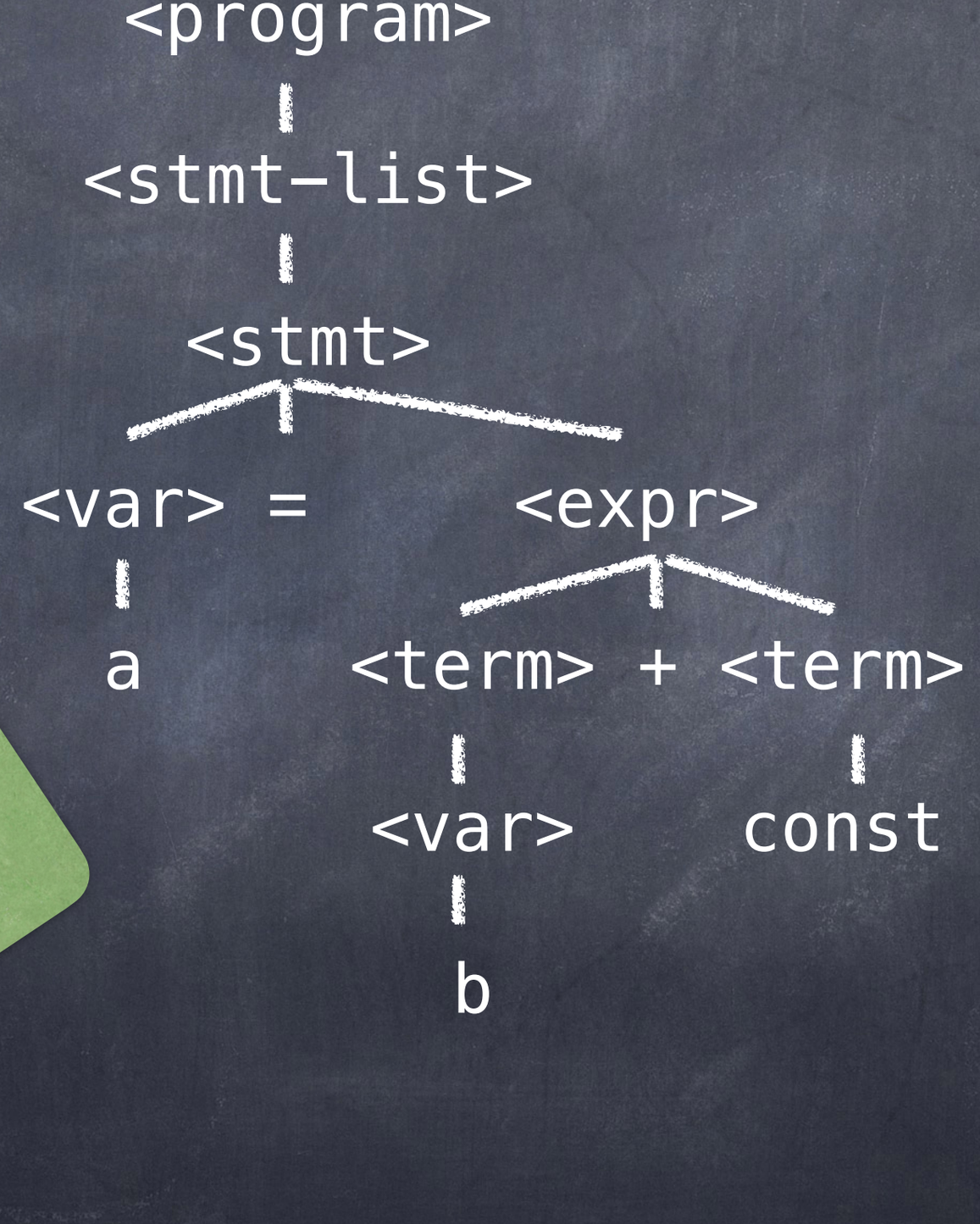
leftmost derivation

```
<program> => <stmt-list>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

rightmost derivation

```
<program> => <stmt-list>
          => <stmt>
          => <var> = <expr>
          => <var> = <term> + <term>
          => <var> = <term> + const
          => <var> = <var> + const
          => <var> = b + const
          => a = b + const
```

Parse tree



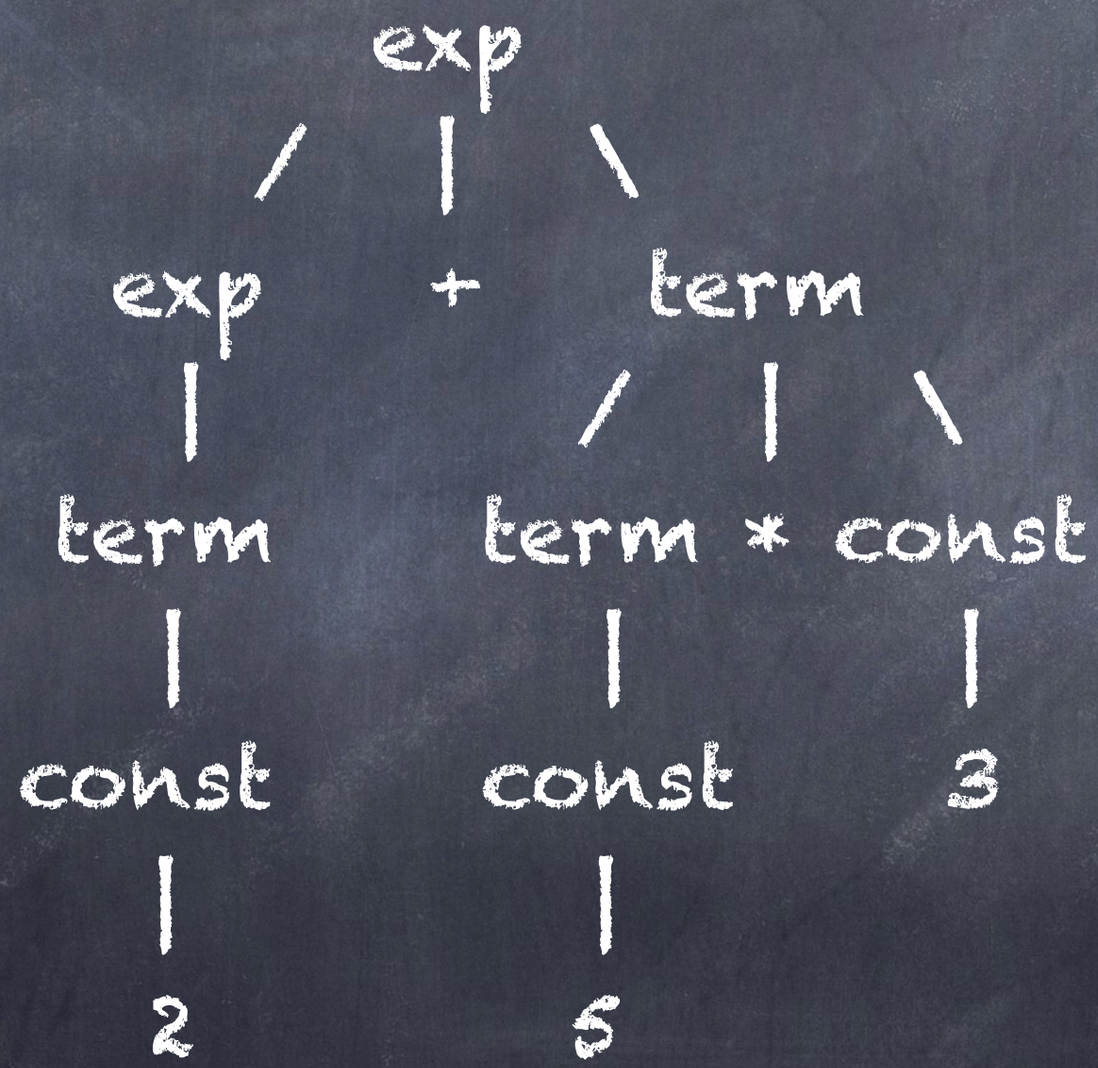
Same parse tree regardless of derivation

Parse trees and compilation

- A compiler builds a parse tree for a program (or for different parts of a program)
- If the compiler cannot build a well-formed parse tree from a given input, it reports a compilation error
- The parse tree serves as the basis for semantic interpretation/translation of the program.

Example

2+5*3



Programming Language Grammar Fragment

```
<program> -> <stmt-list>
<stmt-list> -> <stmt> | <stmt> ; <stmt-list>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const
```

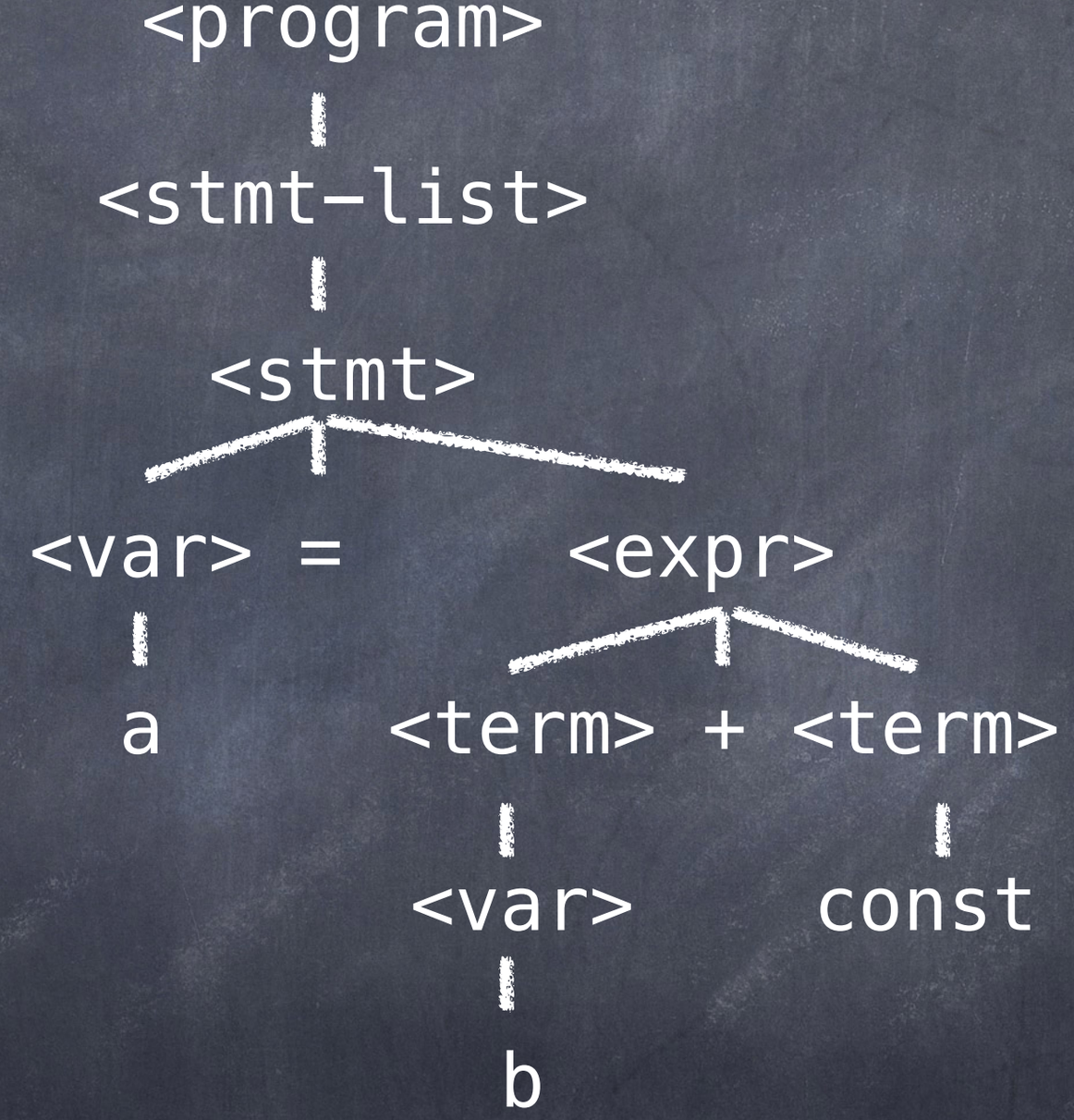
Notes:
<var> is defined in the grammar
const is not defined in the grammar

A leftmost derivation of

a = b + const

```
<program> => <stmt-list>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Parse tree



Parse trees and compilation

- A compiler builds a parse tree for a program (or for different parts of a program)
- If the compiler cannot build a well-formed parse tree from a given input, it reports a compilation error
- The parse tree serves as the basis for semantic interpretation/translation of the program.

Ambiguity

Ambiguity in grammars

- A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.
- Operator precedence and operator associativity are two examples of ways in which a grammar can provide unambiguous interpretation.

Operator precedence ambiguity

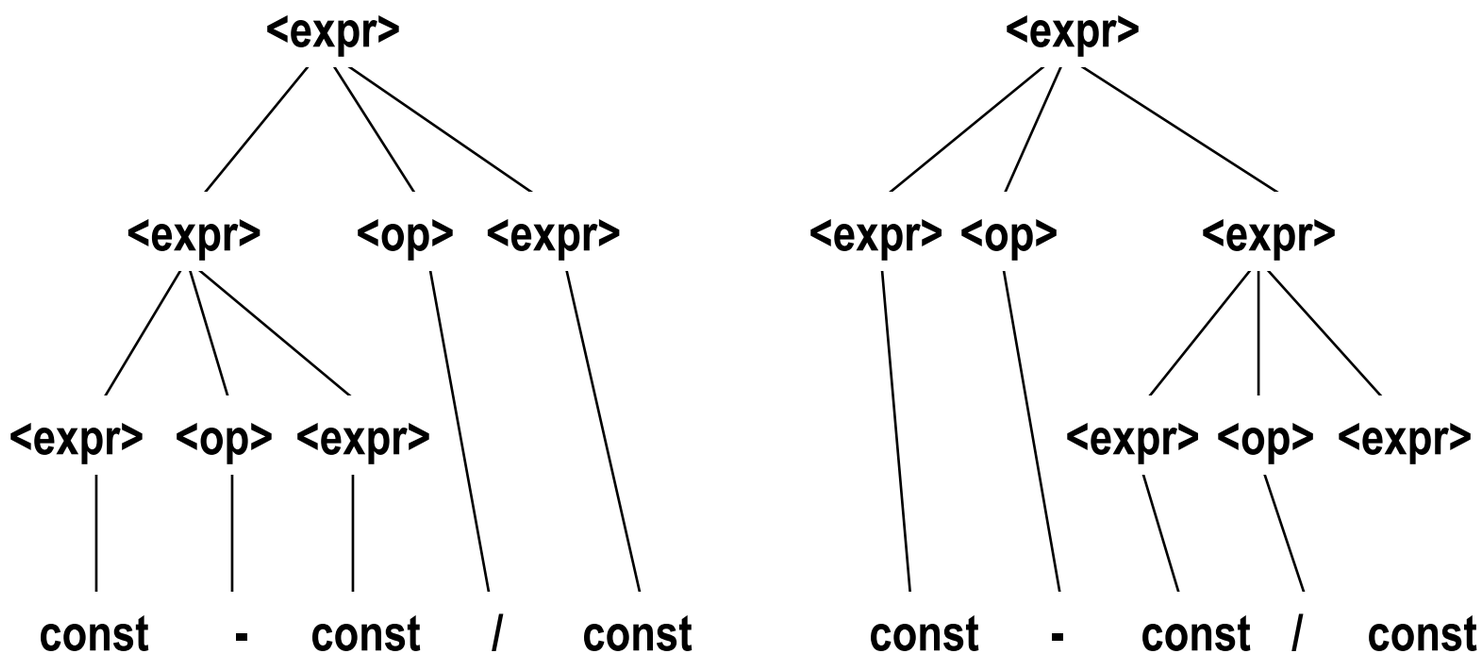
The following grammar is ambiguous:

```
<expr> -> <expr> <op> <expr> | const
<op> -> - | /
```

The grammar treats the two operators, '-' and '/', equivalently

An ambiguous grammar for arithmetic expressions

```
<expr> -> <expr> <op> <expr>  |  const
<op> -> /  |  -
```



Disambiguating the grammar

This grammar (fragment) is unambiguous:

```
<expr> -> <expr> - <term> | <term>
<term> -> <term> / const | const
```

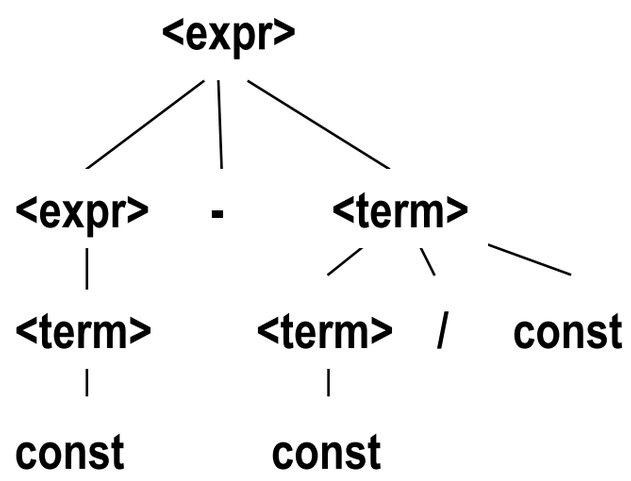
The grammar treats the two operators, '-' and '/', differently.

In this grammar, '/' has higher precedence than '-'. Within a given subtree, deeper nodes are evaluated before shallower notes.

Disambiguating the grammar

- If we use the parse tree to indicate precedence levels of the operators, we can remove the ambiguity.
- The following rules give / a higher precedence than -

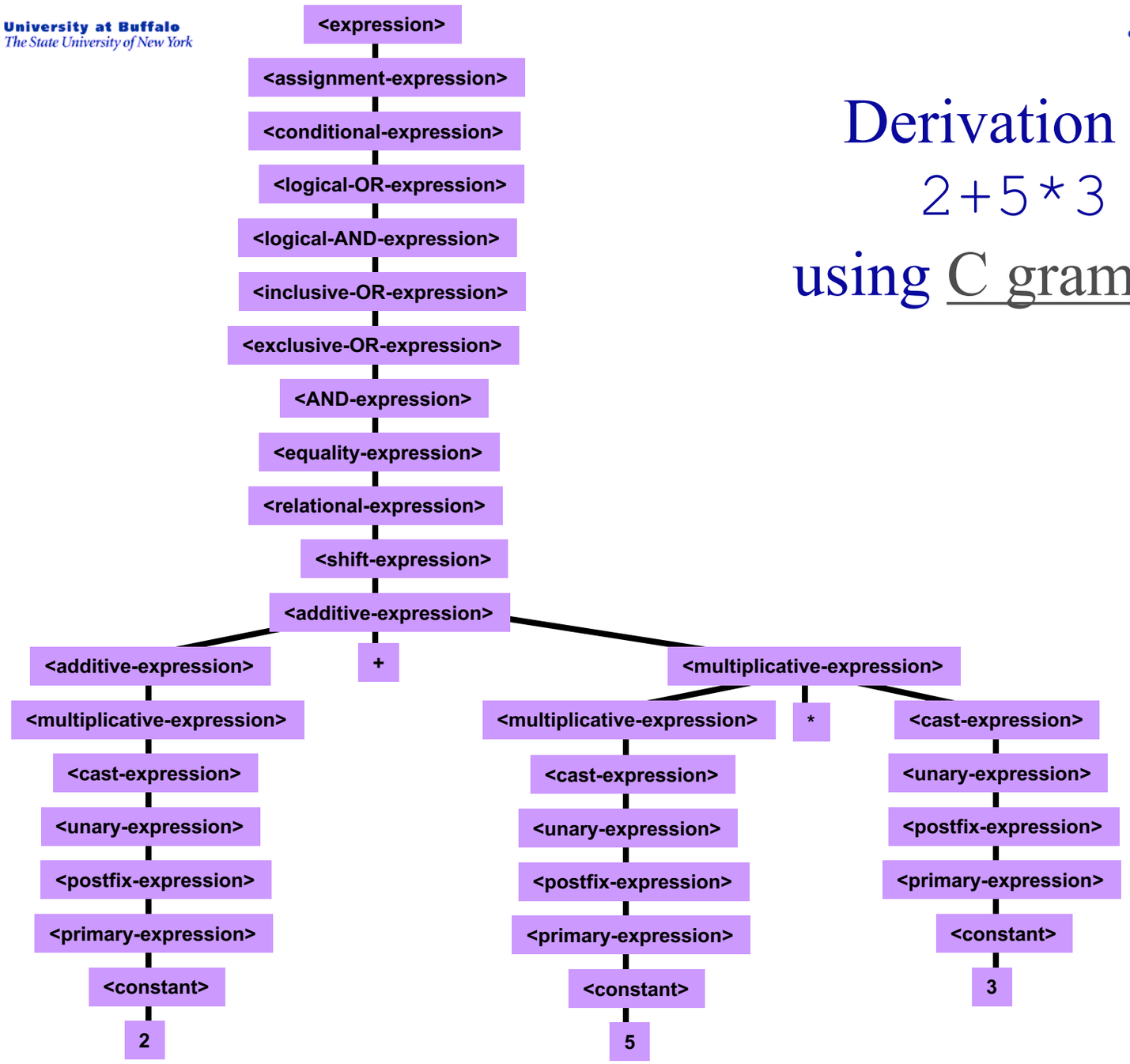
```
<expr> -> <expr> - <term>  |  <term>
<term> -> <term> / const | const
```



University at Buffalo
The State University of New York

cse@buffalo

Derivation of 2+5*3 using C grammar



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>

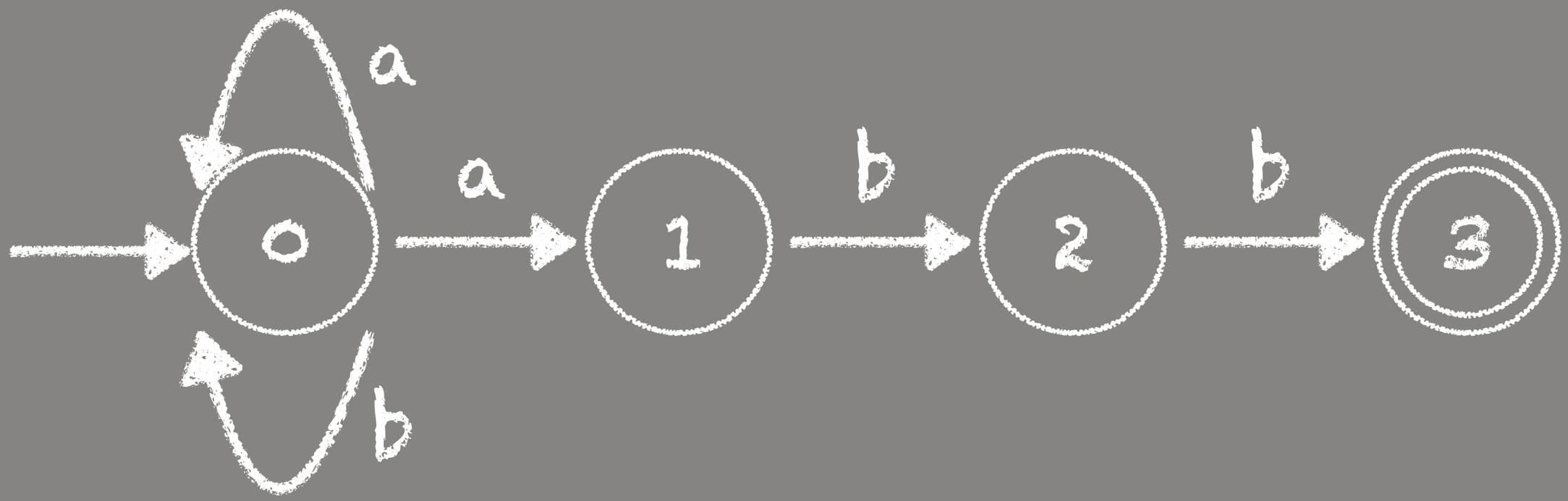
  <term> → <term> * <factor> | <factor>

  <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 $M = (S,\Sigma,\delta,F,s_0)$ 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 $N_i \rightarrow a\ N_j$ to P
  - If $i \in F$, then add $N_i \rightarrow \varepsilon$ to P
  - $S_0 = N_{s0}$

(a|b)*abb



$G = (\{A_0, A_1, A_2, A_3\}, \{a, b\}, \{A_0 \rightarrow a\ A_0, A_0 \rightarrow b\ A_0, A_0 \rightarrow a\ A_1, A_1 \rightarrow b\ A_2, A_2 \rightarrow b\ A_3, A_3 \rightarrow \varepsilon\}, A_0 \}$

RL ⊊ 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: $L \in CFL$, $L \notin RL$

RL ⊊ CFL

Proof (sketch):

$L \in CFL$: $S \rightarrow aSb \mid ab$

$L \notin RL$ (by contradiction):

Assume L is regular. In this case there exists a DFA $D=(N,\Sigma,\delta,F,s_0)$ such that $\mathcal{L}(D) = L$.

Let $k = |N|$. Consider a^ib^i, where $i>k$.

Suppose $\delta(s_0, a^i) = s_r$. Since $i>k$, not all of the states between s_0 and s_r are distinct. Hence, there are v and w, $0 \leq v < w \leq k$ such that $s_v = s_w$. In other words, there is a loop.

This DFA can certainly recognize a^ib^i but it can also recognize a^jb^i, where $i \neq j$, by following the loop.

"REGULAR GRAMMARS CANNOT COUNT"

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

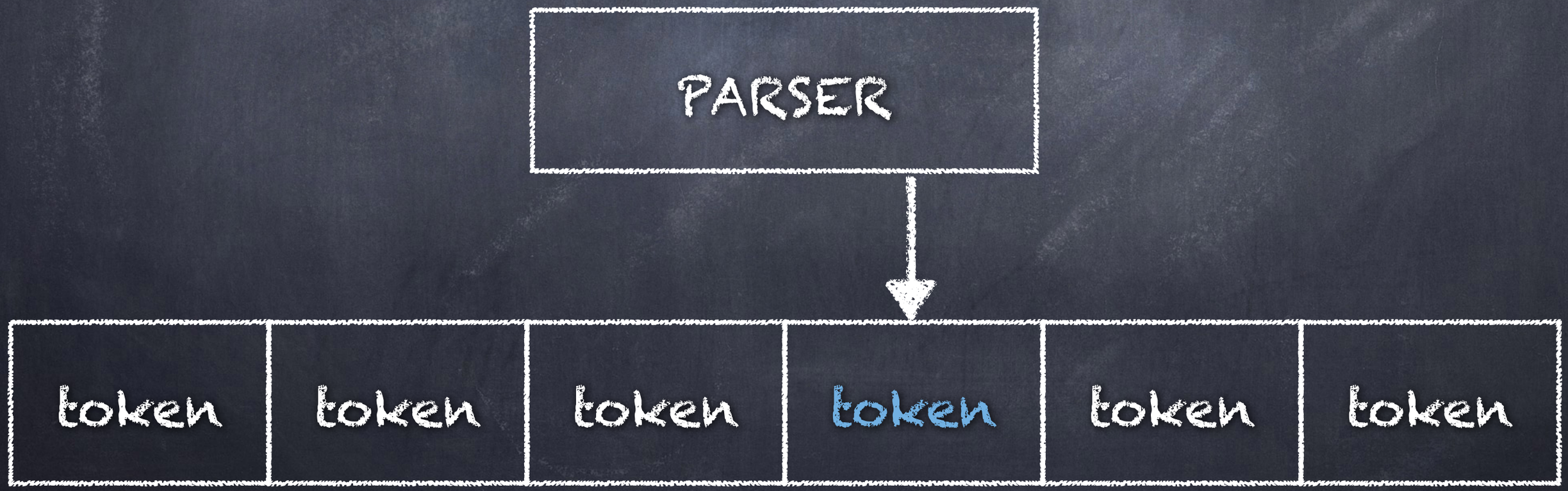
- A top-down parser builds a parse tree from root to the leaves
  - easier to construct by hand
- A bottom-up parser builds a parse tree from leaves to root
  - Handles a larger class of grammars
  - tools (yacc/bison) build bottom-up parsers

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

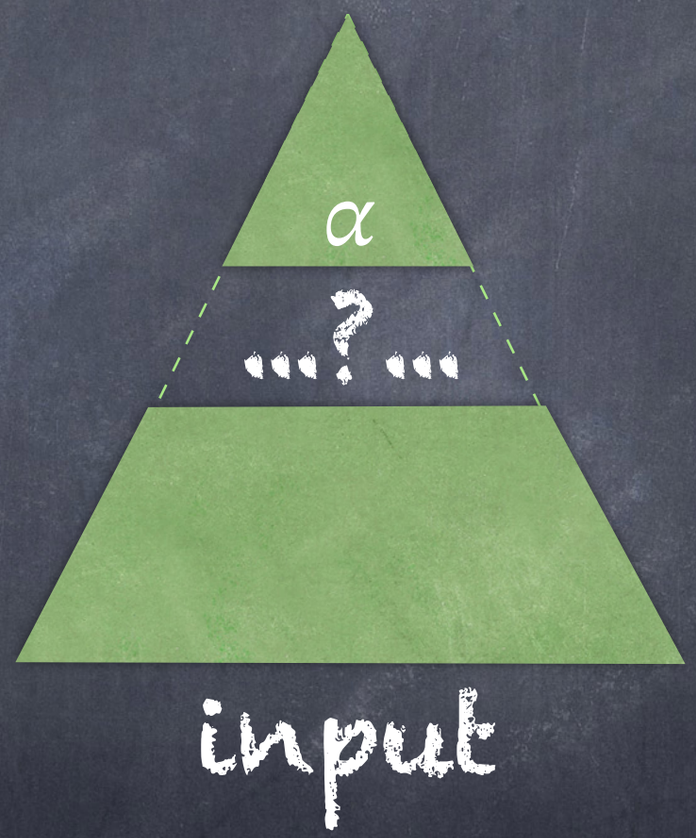
- Start from grammar's start symbol
- Build parse tree so its yield matches input
- predictive parsing: a simple form of recursive descent parsing

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

S =>* α

...?...

=>* input



FIRST(α)

- If α∈(NUT)* 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} ∪ 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 $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$

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.

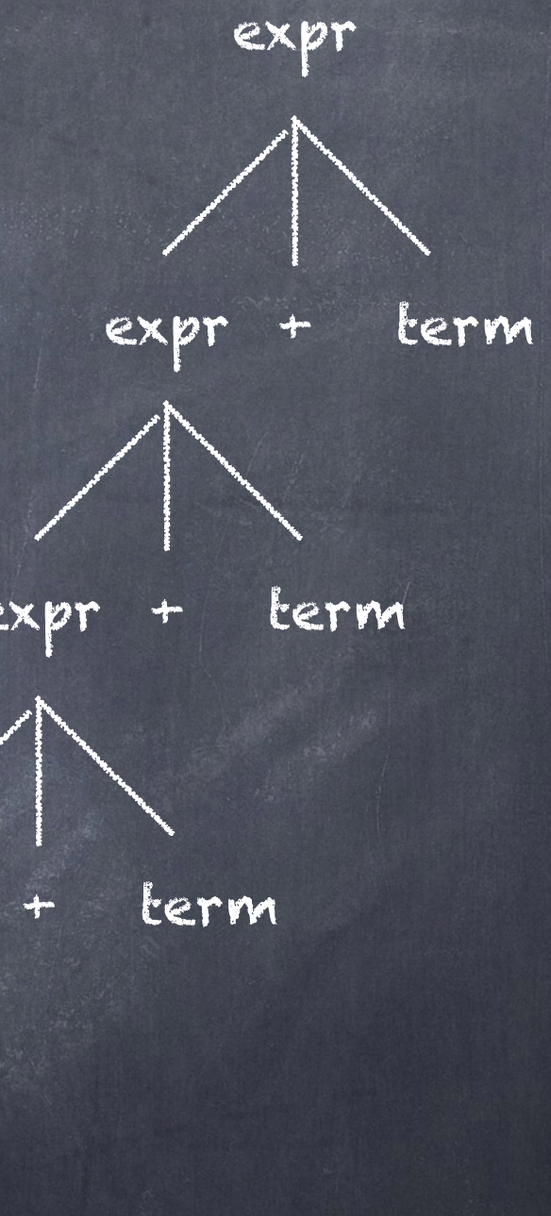
Left recursion example

- Grammar:

  expr -> expr + term | term

  term -> id

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



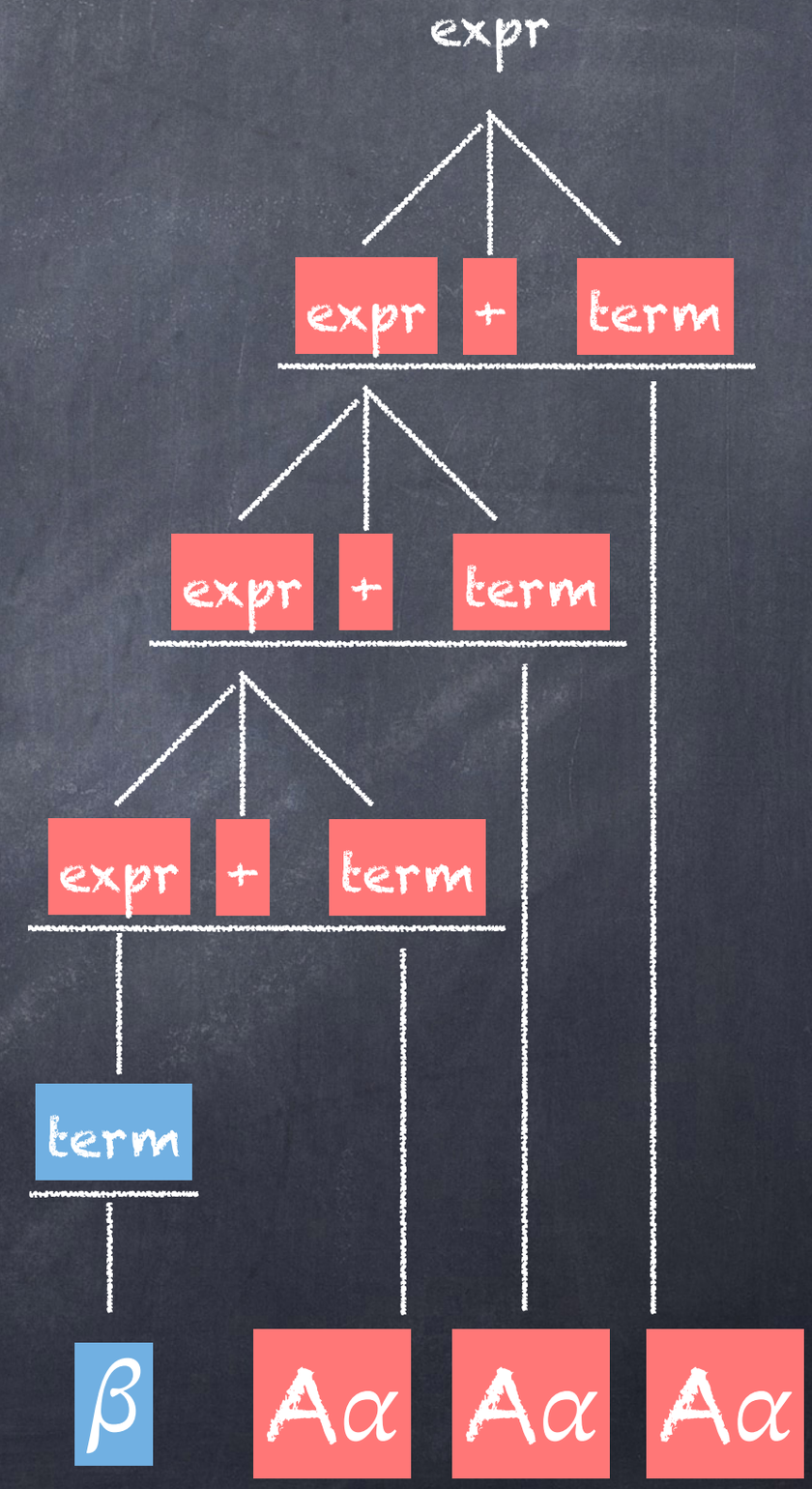
Left recursion example

- Grammar: A α β

  expr -> expr + term | term

  term -> id

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

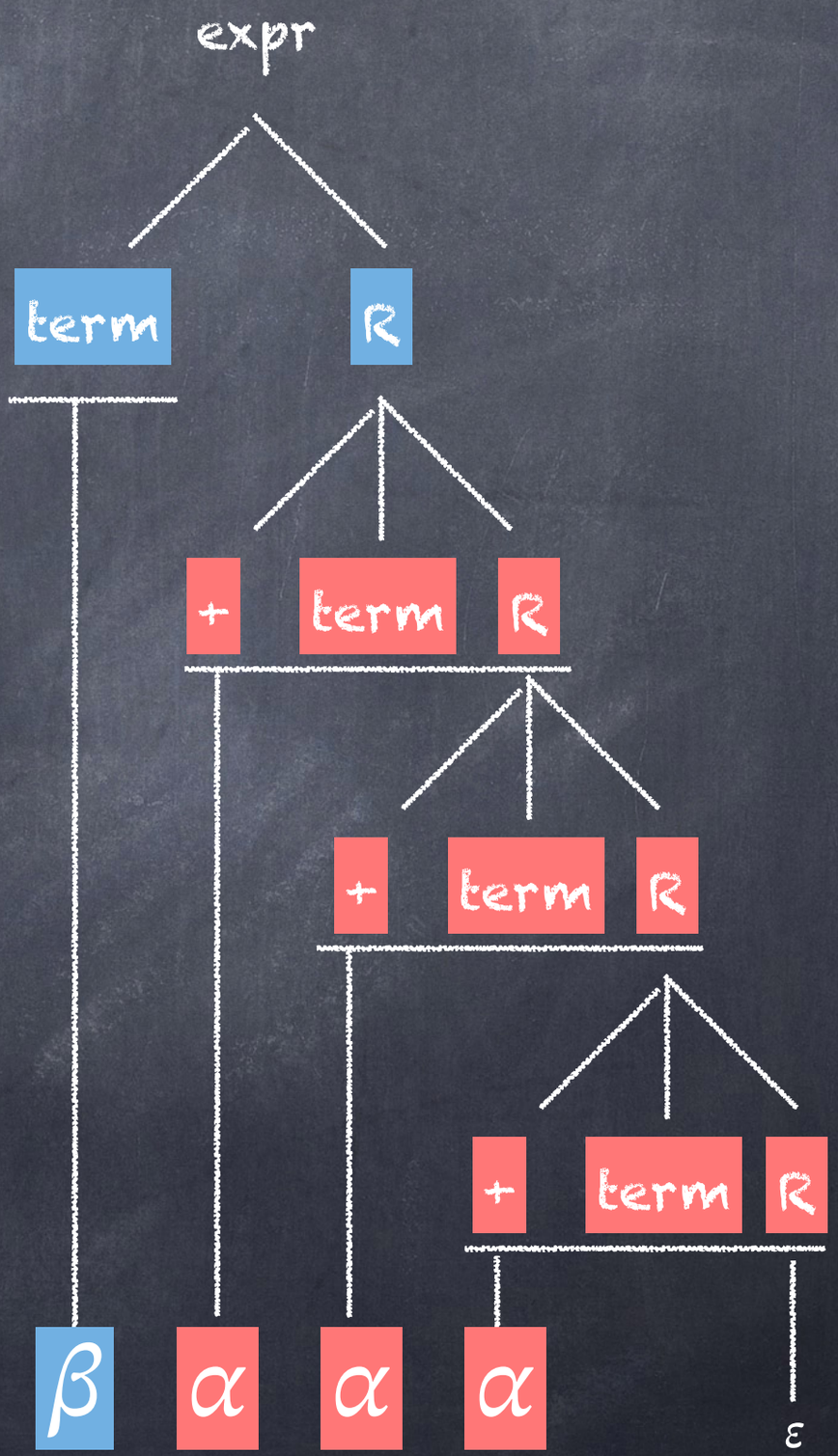


Rewriting grammar to remove left recursion

- 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>