CSE443 Compilers

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

Phases of a compiler

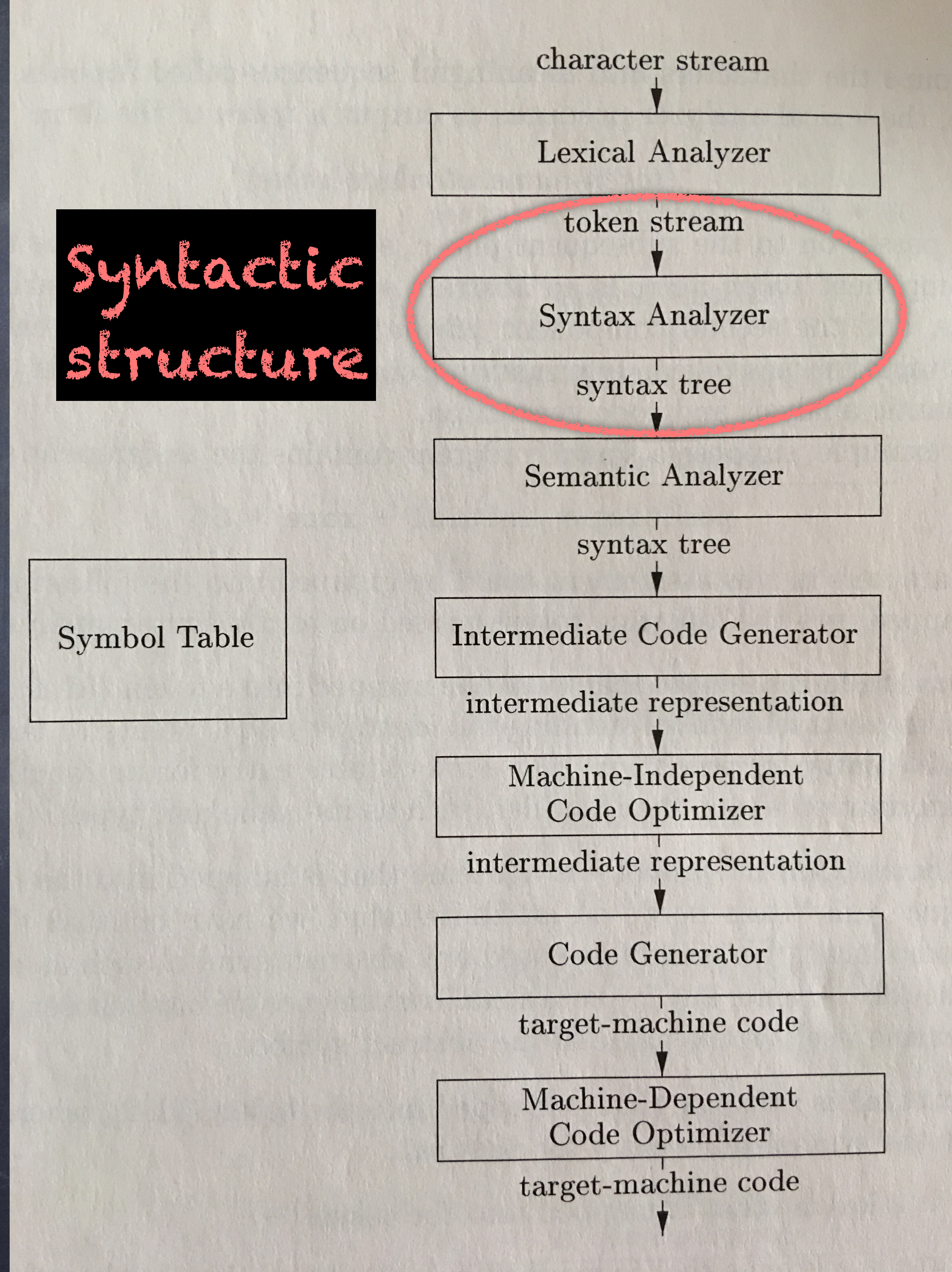


Figure 1.6, page 5 of text

GitHub classroom

- Accept assignment (link in Piazza) to create repo

Part 1 of project

- Four files likely:
  - Makefile
  - lexicalStructure.lex (input to FLEX)
  - typedefs.h (definitions of token values)
  - runner.c (containing a main that calls yylex())
- This structure will change slightly for part 2

- Makefile
  - targets: lex.yy.c, lexer, clean
- lexicalStructure.lex (input to FLEX)
  - regular expressions + supporting code
- typedefs.h (definitions of token values)

  ```
  #define ID 101
  ```
- runner.c (containing a main that calls yylex())

Readings

- Chapter 2 gives good overview of the compilation process.
- Chapters 3 through 9 give details.
- Follow along in textbook as we go though topics, and use it as a reference for details as you work through the project.

Lexical Analysis

- Chapter 3
  - 3.5 discusses the LEX tool. Read the FLEX manual as that's the tool you'll be using.
  - 3.6 and 3.7 go into more detail on NFA to DFA conversion.

Syntax Analysis

- Chapters 4 and 5
  - We'll take a fair bit of time working through this material
  - Consult text on an as-needed basis for details
  - 4.9 discusses the YACC tool. Read the BISON manual as that's the tool you'll be using.

Language terminology

(from Sebesta (10th ed), p. 115)

- A *language* is a set of strings of symbols, drawn from some finite set of symbols (called the alphabet of the language).
- "The strings of a language are called **sentences**"
- "Formal descriptions of the syntax […] do not include descriptions of the lowest-level syntactic units […] called **lexemes**."
- "A **token** of a language is a category of its lexemes."
- Syntax of a programming language is often presented in two parts:
  - regular grammar for token structure (e.g. structure of identifiers)
  - context-free grammar for sentence structure

Examples of *lexemes* and *tokens*

Lexemes	***Tokens***
foo	identifier
i	identifier
sum	identifier
-3	integer_literal
10	integer_literal
1	integer_literal
;	statement_separator
=	assignment_operator

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58, modified by Peter Naur for ALGOL 60
  - BNF is equivalent to context-free grammar
  - BNF is a *metalanguage* used to describe another language, the *object language*
  - Extended BNF: adds syntactic sugar to produce more readable descriptions

BNF Fundamentals

- Sample rules [p. 128]

    <assign> → <var> = <expression>
    <if_stmt> → **if** <logic_expr> **then** <stmt>
    <if_stmt> → **if** <logic_expr> **then** <stmt> **else** <stmt>

- non-terminals surrounded by < and >
- tokens are not surrounded by < and >
- keywords in language are in **bold**
- → separates LHS from RHS
- | expresses alternative expansions for LHS

    <if_stmt> → **if** <logic_expr> **then** <stmt>
    | **if** <logic_expr> **then** <stmt> **else** <stmt>

- = is in this example a singleton token represented by its sole lexeme

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- A grammar is often given simply as a set of rules (terminal and non-terminal sets are implicit in rules, as is start symbol)

Describing Lists

- There are many situations in which a programming language allows a list of items (e.g. parameter list, argument list).
- Such a list can typically be as short as empty or consisting of one item.
- Such lists are typically not bounded.
- How is their structure described?

Describing lists

- The are described using ***recursive rules***.
- Here is a pair of rules describing a list of identifiers, whose minimum length is one:

```
<ident_list> -> ident
              | ident , <ident_list>
```

- Notice that '`,`' is part of the *object language* (the language being described by the grammar).

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

Observations

- Every string of symbols in a derivation is a sentential form.
- A sentence is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation can be leftmost, rightmost, or neither.

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
```

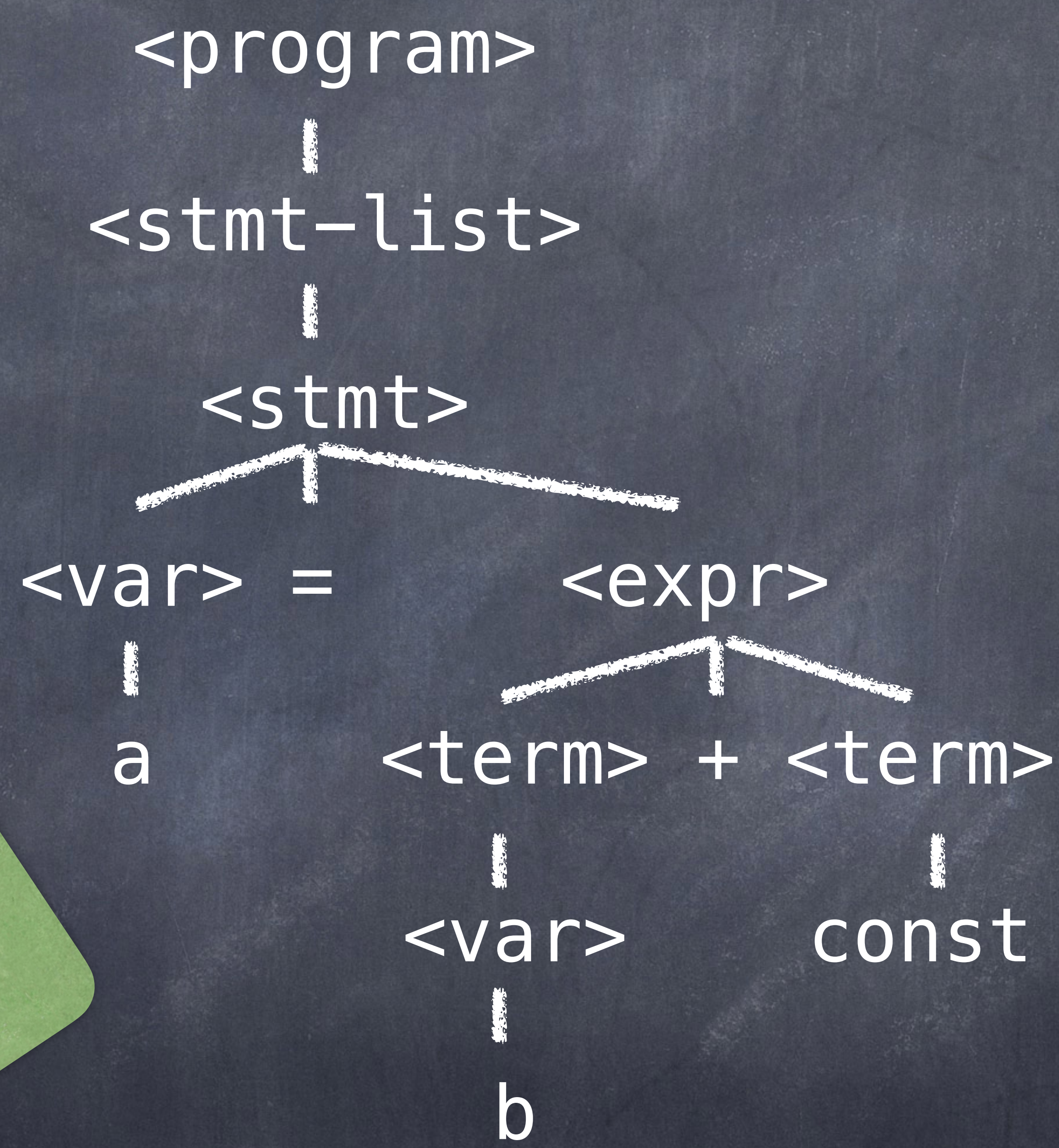
leftmost derivation

```
<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.

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

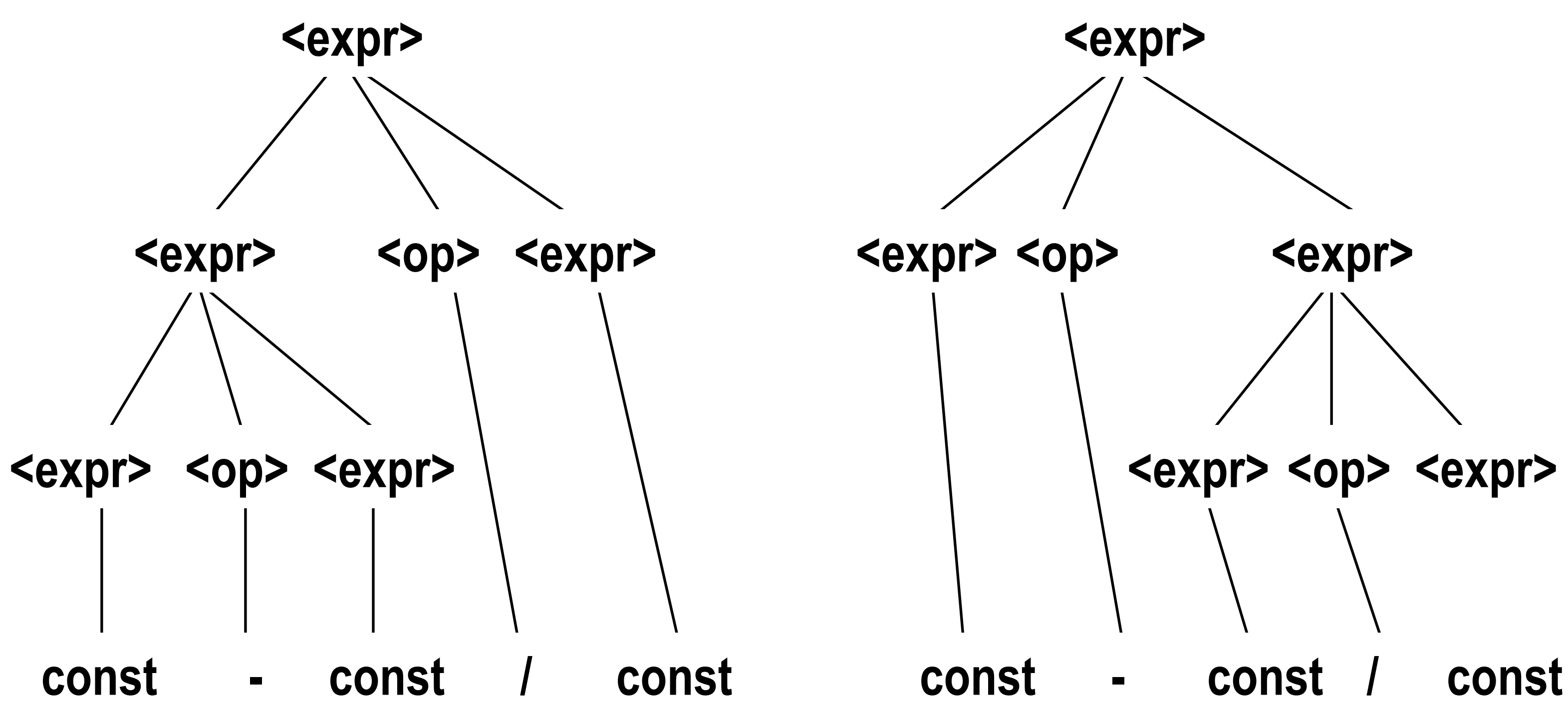
The following grammar is ambiguous:

```
<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
```

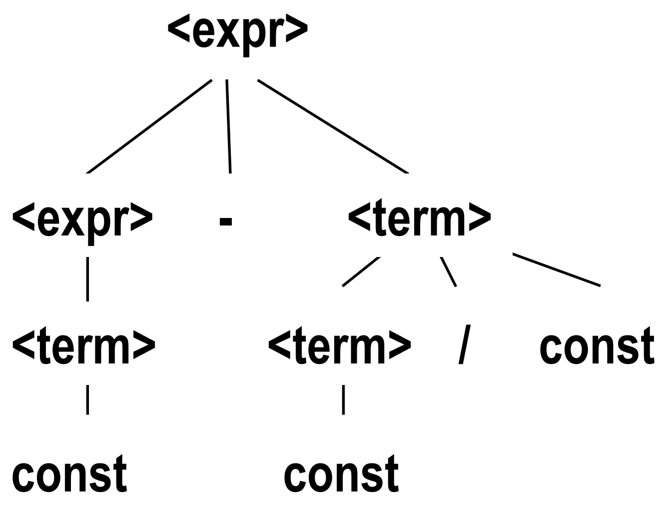
The grammar treats the two operators, '-' and '/', differently.

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
```



Derivation of 2+5*3 using C grammar