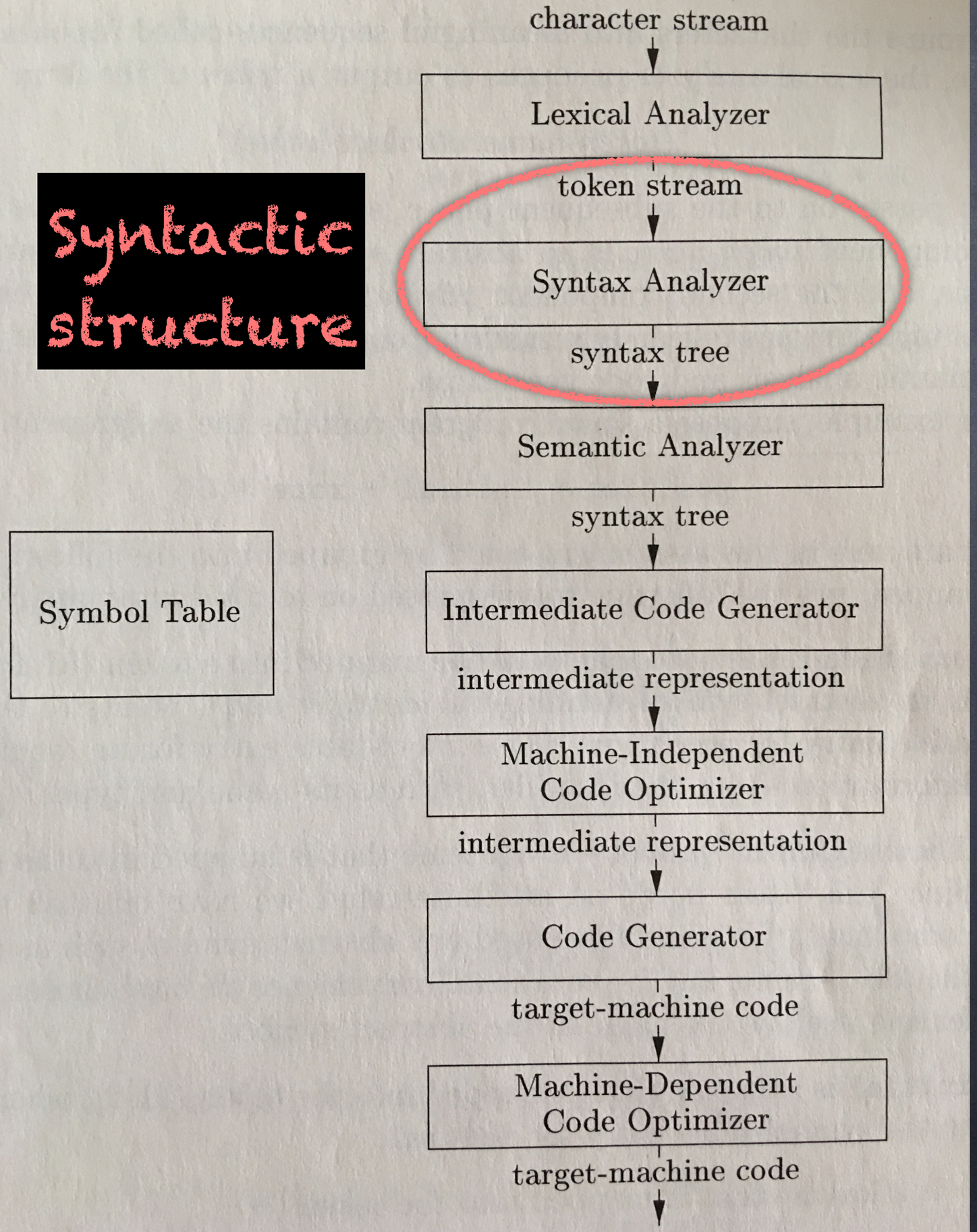
CSE443
Compilers

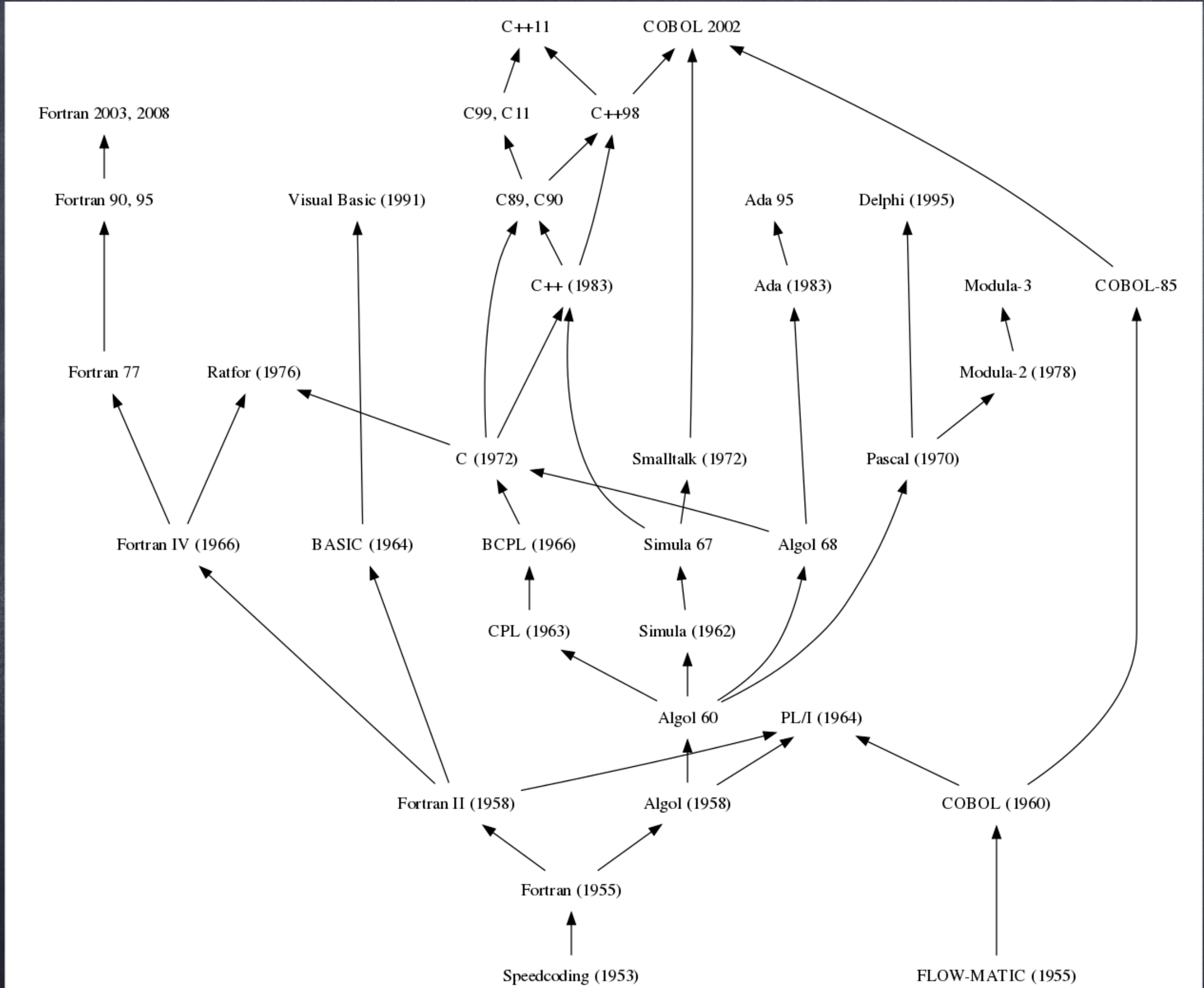
Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

Figure 1.6, page 5 of text

Syntactic structure





http://www.softwarepreservation.org/projects/FORTRAN/paper/p4-backus.pdf

http://www.bitsavers.org/pdf/univac/flow-matic/U1518_FLOW-MATIC_Programming_System_1958.pdf

Rear Admiral Grace Murray Hopper
(1906 - 1992)

In 1952, Hopper completed her first compiler (for Sperry-Rand computer), known as the *A-0 System*. [...]

After the A-0, Grace Hopper and her group produced versions A-1 and A-2, improvements over the older version. The A-2 compiler was the first compiler to be used extensively, paving the way to the development of programming languages.

[...]

Hopper also originated the idea that computer programs could be written in English. She viewed letters as simply another kind of symbol that the computer could recognize and convert into machine code. Hopper's compiler later evolved to FLOW-MATIC compiler, which will be the base for the extremely important language—COBOL.

https://history-computer.com/ModernComputer/Software/FirstCompiler.html

Context Free Grammars

CFG $G = (N, T, P, S)$

N is a set of non-terminals

T is a set of terminals (= tokens from lexical analyzer)

$T \cap N = \varnothing$ (i.e. a symbol is either a terminal or a non-terminal, not both)

P is a set of productions/grammar rules

$P \subseteq N \times (N \cup T)^*$

$R \in P$ is written as $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^*$

$S \in N$ is the start symbol

Derivations

$\Rightarrow_G$ "derives in one step (from G)"

If $A \to \beta \in P$, and $\alpha, \gamma \in (N \cup T)^*$ then $\alpha A \gamma \Rightarrow_G \alpha \beta \gamma$

$\Rightarrow_G^*$ "derives in many steps (from G)"

If $\alpha_i \in (N \cup T)^*$, $m \geq 1$ and $\alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \alpha_3 \Rightarrow_G \alpha_4 \ \ldots \Rightarrow_G \alpha_m$

then $\alpha_1 \Rightarrow_G^* \alpha_m$

$\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$

Languages

$$\mathcal{L}(G) = \{ w \mid w \in T^* \text{ and } S \Rightarrow_G^* w \}$$

L is a CF language if it is $\mathcal{L}(G)$ for a CFG G.

G1 and G2 are equivalent iff $\mathcal{L}(G1)=\mathcal{L}(G2)$.

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).

Derivation of sentences from a grammar

- A ***derivation*** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

G_2 = ({a, the, dog, cat, chased},
{S, NP, VP, Det, N, V},
{S → NP VP, NP → Det N, Det → a | the,
N → dog | cat, VP → V | VP NP, V → chased},
S)

Example: derivation from G_2

- Example: derivation of *the dog chased a cat*

S => NP VP
=> Det N VP
=> the N VP
=> the dog VP
=> the dog V NP
=> the dog chased NP
=> the dog chased Det N
=> the dog chased a N
=> the dog chased a cat

Example

L = { 0, 1, 00, 11, 000, 111, 0000, 1111, ... }

G = ({0,1}, {S, ZeroList, OneList},
{S -> ZeroList | OneList,
ZeroList -> 0 | 0 ZeroList,
OneList -> 1 | 1 OneList },
S)

Derivations from G

Derivation of 0 0 0 0

S => ZeroList
=> 0 ZeroList
=> 0 0 ZeroList
=> 0 0 0 ZeroList
=> 0 0 0 0

Derivation of 1 1 1

S => OneList
=> 1 OneList
=> 1 1 OneList
=> 1 1 1

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
```

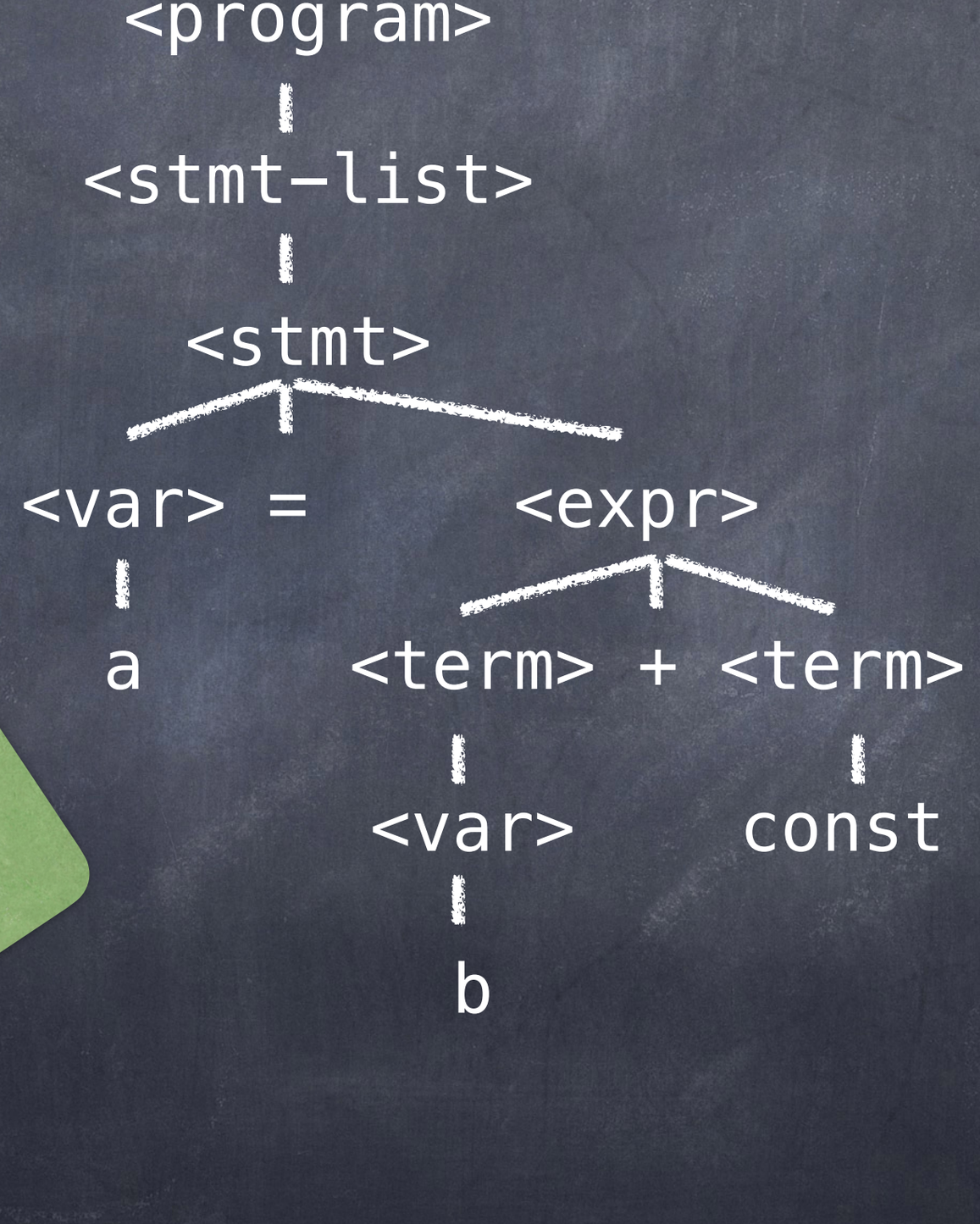
leftmost derivation

```
<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



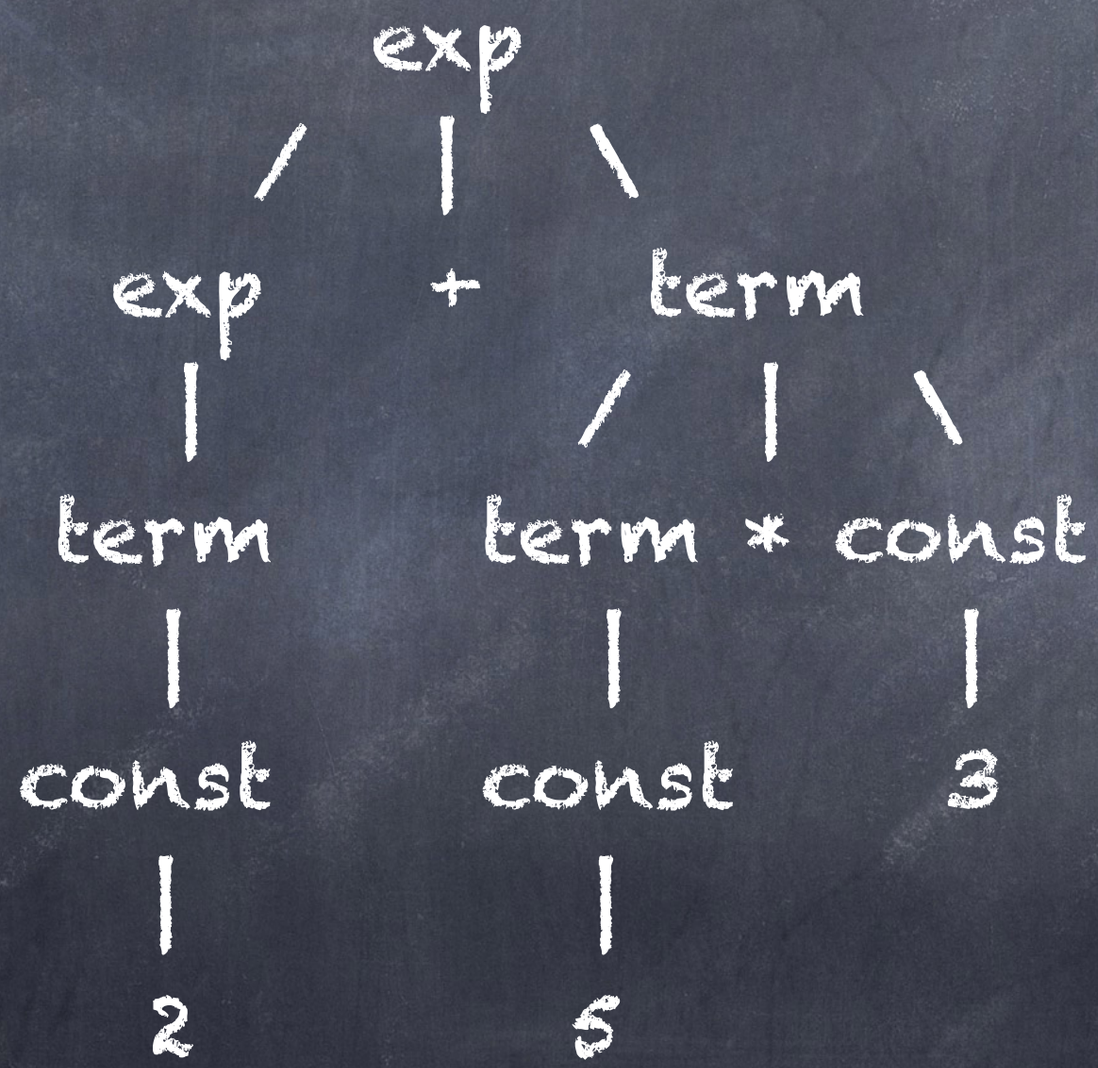
Same parse tree regardless of derivation

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.

Example

2+5*3



University at Buffalo
The State University of New York

cse@buffalo

Derivation of 2+5*3 using C grammar