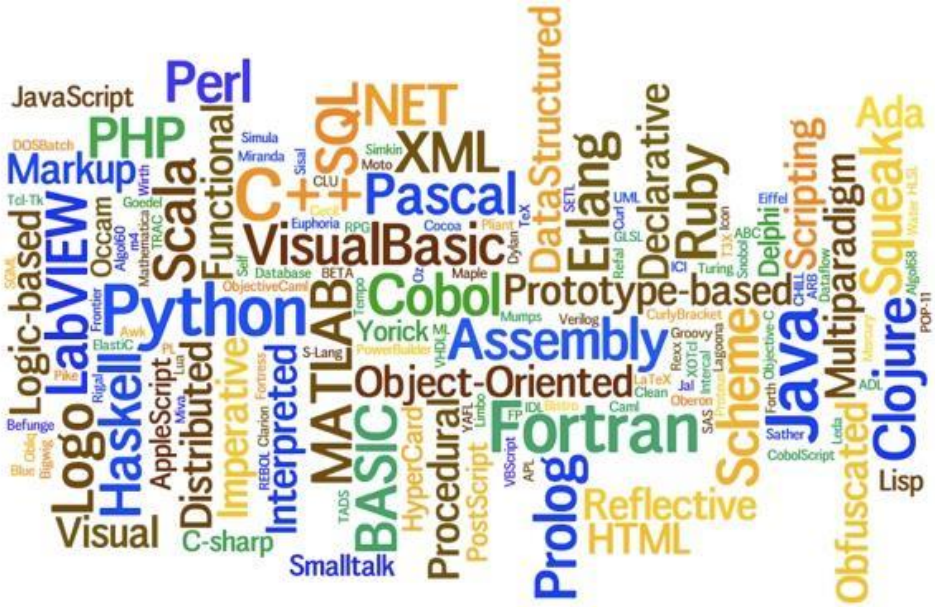


Programming Languages Syntax I

Theory of Programming Languages
Module 2

Dr. Tamer ABUHMED
College of Computing

SUNG KYUN KWAN UNIVERSITY

Outline

- Introduction (**Chapter 3 of the textbook**)
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax

Textbook chapters

Table of Contents

Introduction

- For the next several weeks we'll look at how one can define a programming language
- What is a language, anyway?

  "Language is a system of gestures, grammar, signs, sounds, symbols, or words, which is used to represent and communicate concepts, ideas, meanings, and thoughts"
- Human language is a way to communicate representations from one (human) mind to another
- What about a programming language?

  A way to communicate representations (e.g., of data or a procedure) between human minds and/or machines

Introduction Cont.

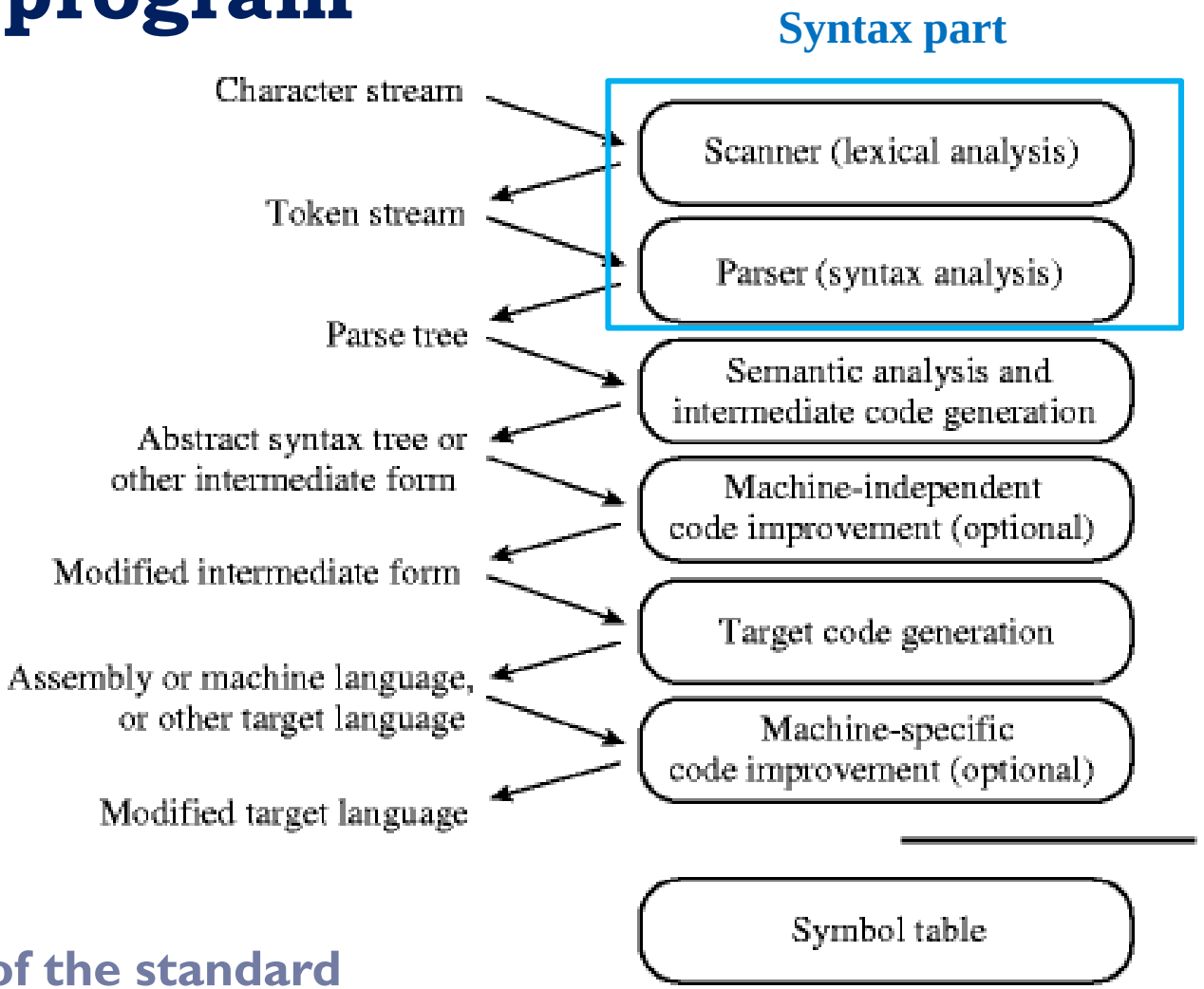
- We usually break down the problem of *defining* a programming language into two parts
  - defining the PL's syntax
  - defining the PL's semantics
  - *Syntax* - the form or structure of the expressions, statements, and program units
  - *Semantics* - the meaning of the expressions, statements, and program units
- Note: There is not always a clear boundary between the two

Why and How

- Why do we need syntax and semantics?
  - We want specifications for several communities:
  - Other language designers
  - Implementers
  - Machines?
  - Programmers (the users of the language)

- How? One ways is via natural language descriptions (e.g., user's manuals, text books) but there are a number of more formal techniques for specifying the syntax and semantics

Source to program



This is an overview of the standard process of turning a text file into an executable program.

Syntax Overview

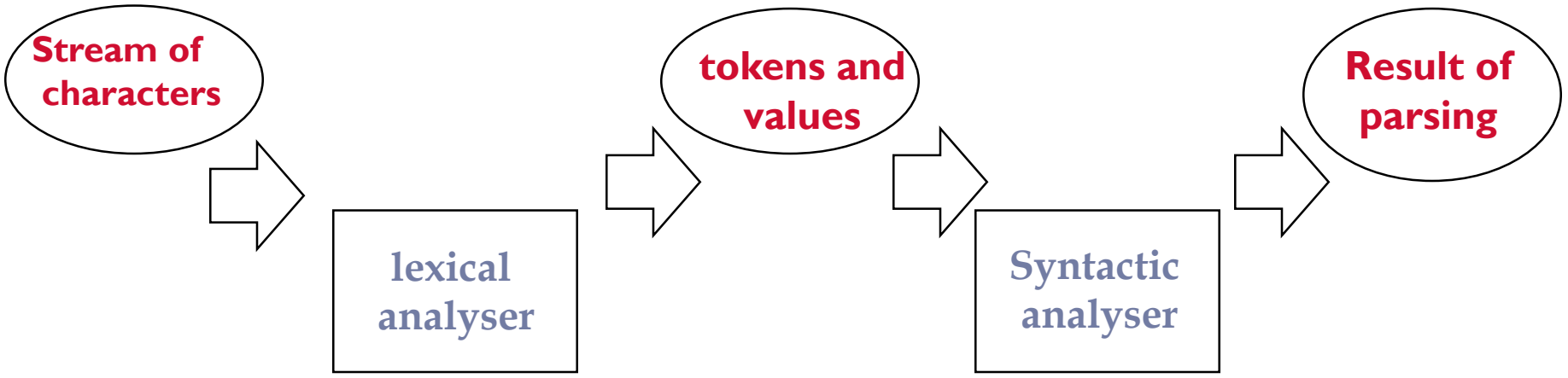
- Language preliminaries
- Context-free grammars and BNF
- Syntax diagrams

Preliminaries

- A *sentence* is a string of characters over some alphabet (e.g., *def add1(n): return n + 1*)
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, *add1*, *begin*)
- A *token* is a category of lexemes (e.g., *identifier*)
- Formal approaches to describing syntax:
  - Recognizers - device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Generators - A device that generates sentences of a language. One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

Lexical Structure of Programming Languages

- The structure of its lexemes (words or tokens)
  - token is a category of lexeme
- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase (syntactic analyser) determines syntactic structure



Formal Grammar

- A (formal) grammar is a set of rules for strings in a formal language
- The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax
- A grammar does not describe the meaning of the strings or what can be done with them in whatever context — only their form

Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s.
  - Language generators, meant to describe the syntax of natural languages.
  - Define a class of languages called *context-free languages*.

- Backus Normal/Naur Form (1959)
  - Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
  - BNF is equivalent to context-free grammars


Noam Chomsky

- Chomsky & Backus independently came up with equivalent formalisms for specifying the syntax of a language
- Backus focused on a practical way of specifying an artificial language, like Algol
- Chomsky made fundamental contributions to mathematical linguistics and was motivated by the study of human languages.

NOAM CHOMSKY, MIT Institute Professor; Professor of Linguistics, Linguistic Theory, Syntax, Semantics, Philosophy of Language

Six participants in the 1960 Algol conference in Paris. This was taken at the 1974 ACM conference on the history of programming languages. Top: John McCarthy, Fritz Bauer, Joe Wegstein. Bottom: *John Backus, Peter Naur,* Alan Perlis.

BNF

- A rule has a left-hand side (LHS) which is a single *non-terminal* symbol and a right-hand side (RHS), one or more *terminal* or *non-terminal* symbols
- A *grammar* is a finite, nonempty set of rules
- A *non-terminal* symbol is "defined" by its rules.
- Multiple rules can be combined with the vertical-bar (|) symbol (read as "or")
- These two rules:
  - `<stmts> ::= <stmt>`
  - `<stmts> ::= <stmnt> ; <stmnts>`
- are equivalent to this one:
  - `<stmts> ::= <stmt> | <stmnt> ; <stmnts>`

BNF (continued)

- A *metalanguage* is a language used to describe another language.
- In BNF, *abstractions* are used to represent classes of syntactic structures -- they act like syntactic variables (also called nonterminal *symbols*), e.g.
  - `<while_stmt> ::= while <logic_expr> do <stmt>`
- This is a *rule*; it describes the structure of a while statement

Non-terminals, pre-terminals & terminals

- A non-terminal symbol is any symbol that is in the LHS of a rule. These represent abstractions in the language (e.g., *if-then-else-statement* in

```
<if-then-else-statement> ::= if <test>
  then <statement> else <statement>
```

- A terminal symbol is any symbol that is not on the LHS of a rule. AKA *lexemes*. These are the literal symbols that will appear in a program (e.g., *if*, *then*, *else* in rules above).
- A pre-terminal symbol is one that appears as a LHS of rule(s), but in every case, the RHSs consist of single terminal symbol, e.g., <digit> in

```
<digit> ::= 0 | 1 | 2 | 3 … 7 | 8 | 9
```

BNF

- Repetition is done with recursion
- E.g., Syntactic lists are described in BNF using recursion
- An <ident_list> is a sequence of one or more <ident>s separated by commas.

```
<ident_list> ::= <ident> |
                 <ident> , <ident_list>
```

BNF Example

- Here is an example of a simple grammar for a subset of English
- A sentence is noun phrase and verb phrase followed by a period.

<sentence> ::= <nounPhrase> <verbPhrase> .

<nounPhrase> ::= <article> <noun>

<article> ::= a | the

<noun> ::= man | apple | worm | penguin

<verbPhrase> ::= <verb>|<verb> <nounPhrase>

<verb> ::= eats | throws | sees | is

Derivations

- A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence consisting of just all terminal symbols
- It demonstrates, or proves that the derived sentence is "generated" by the grammar and is thus in the language that the grammar defines
- As an example, consider our baby English grammar

```
<sentence>     ::= <nounPhrase><verbPhrase>.
<nounPhrase>   ::= <article><noun>
<article>      ::= a | the
<noun>         ::= man | apple | worm | penguin
<verbPhrase>   ::= <verb> | <verb><nounPhrase>
<verb>         ::= eats | throws | sees | is
```

Derivation using BNF

Here is a derivation for "the man eats the apple."

<sentence> → <nounPhrase><verbPhrase>.

<article><noun><verbPhrase>.

the<noun><verbPhrase>.

the man <verbPhrase>.

the man <verb><nounPhrase>.

the man eats <nounPhrase>.

the man eats <article> < noun>.

the man eats the <noun>.

the man eats the apple.

Derivation

- Every string of symbols in the 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 in the next step
- A derivation may be either leftmost or rightmost or something else

Another BNF Example

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

Note: There is some variation in notation for BNF grammars. Here we are using -> in the rules instead of ::= .

- Here is a derivation:

<program> => <stmts>
=> <stmt>
=> <var> = <expr>
=> a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

Finite and Infinite languages

- A simple language may have a finite number of sentences
- The set of strings representing integers between -10**6 and +10**6 is a finite language
- A finite language can be defined by enumerating the sentences, but using a grammar might be much easier
- Most interesting languages have an infinite number of sentences

Is English a finite or infinite language?

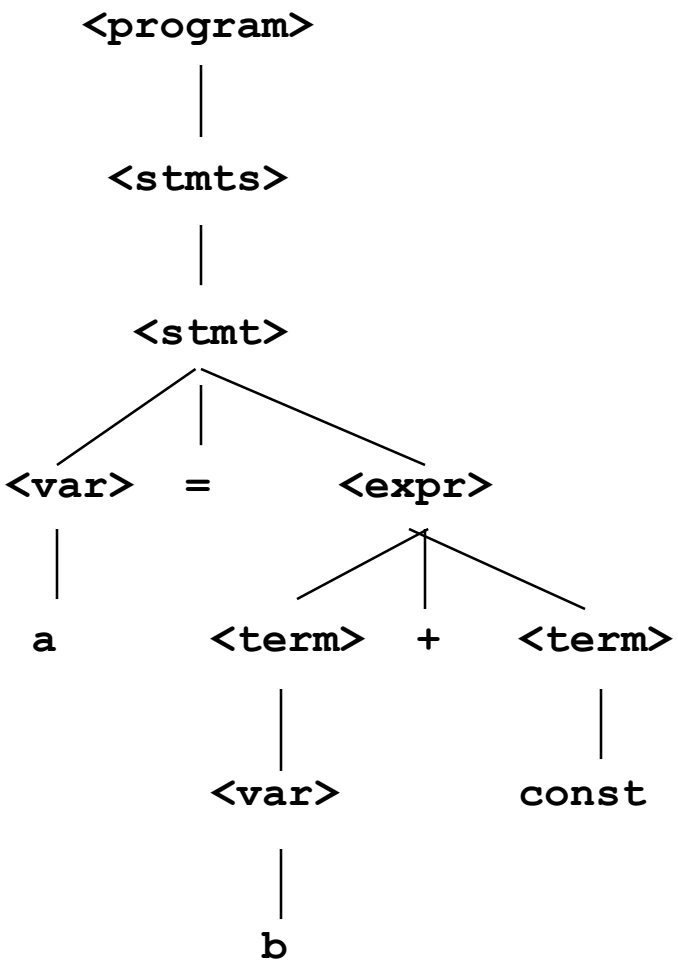
- Assume we have a finite set of words
- Consider adding rules like the following to the previous example

```
<sentence> ::= <sentence><conj><sentence>.
<conj>     ::= and | or | because
```

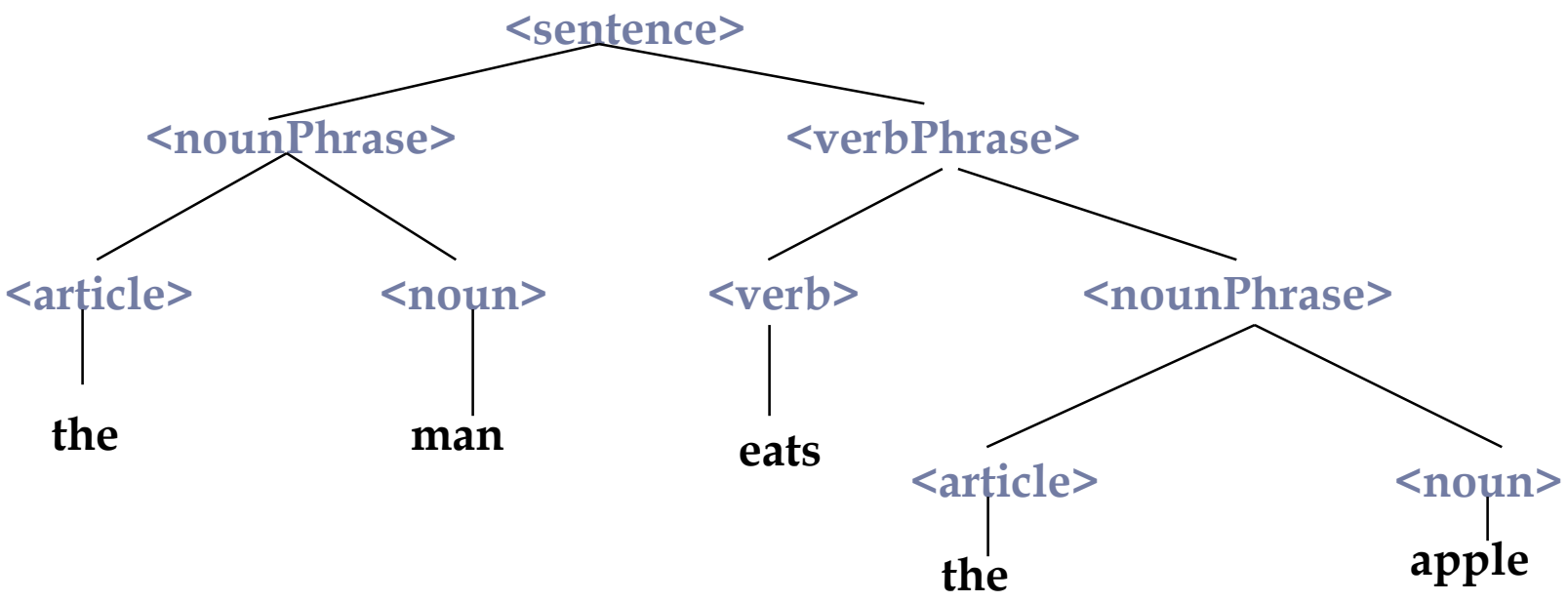
- Hint: Whenever you see recursion in a BNF it's likely that the language is infinite.
  - When might it not be?
  - *The recursive rule might not be reachable. There might be epsilons.*

Parse Tree

A *parse tree* is a hierarchical representation of a derivation



Another Parse Tree



Grammar

- A grammar is ambiguous *if and only if* (iff) it generates a sentential form that has two or more distinct parse trees
- Ambiguous grammars are, in general, very undesirable in *formal languages*
  - Can you guess why?
- We can eliminate ambiguity by revising the grammar

An ambiguous grammar

- Here is a simple grammar for expressions that is ambiguous
  - <e> → <e> <op> <e>
  - <e> → 1|2|3
  - <op> → +|-|*|/

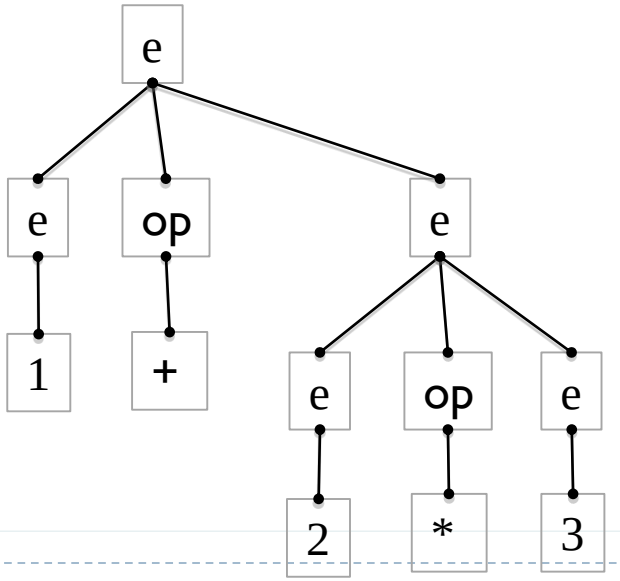
Fyi… In a programming language, an expression is some code that is evaluated and produces a value. A statement is code that is executed and does something but does not produce a value.

- The sentence *1+2*3* can lead to two different parse trees corresponding to *1+(2*3)* and *(1+2)*3*

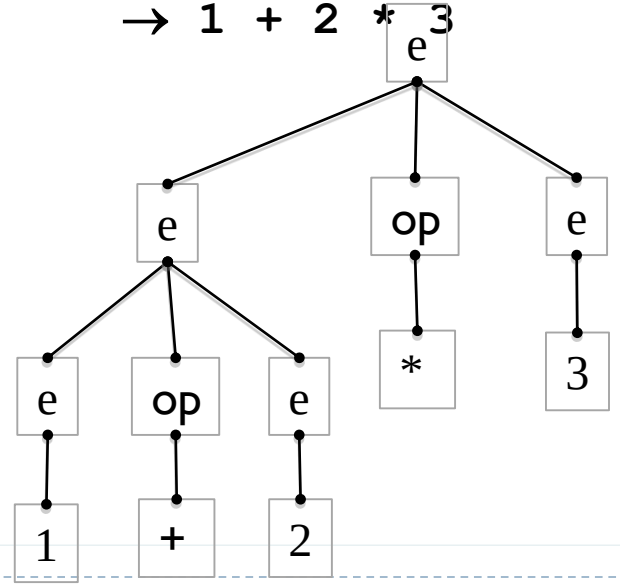
Two derivations for 1+2*3

<e> -> <e> <op> <e>
<e> -> 1|2|3
<op> -> +|-|*|/

<e> → <e> <op> <e>
→ 1 <op> <e>
→ 1 + <e>
→ 1 + <e> <op> <e>
→ 1 + 2 <op> <e>
→ 1 + 2 * <e>
→ 1 + 2 * 3



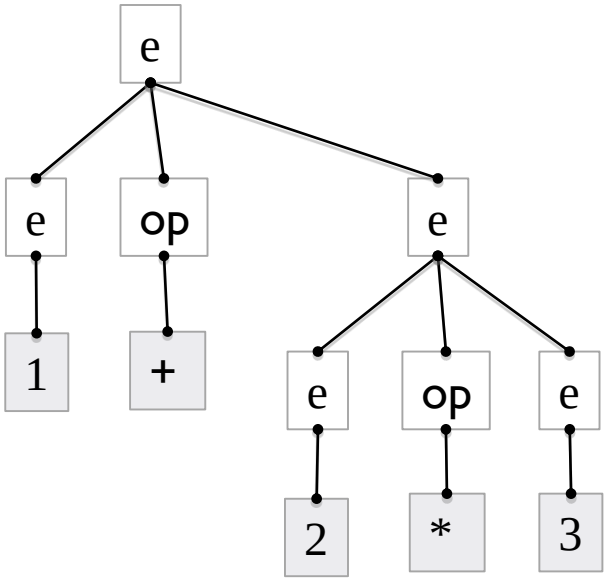
<e> → <e> <op> <e>
→ <e> <op> <e> <op> <e>
→ 1 <op> <e> <op> <e>
→ 1 + <e> <op> <e>
→ 1 + 2 <op> <e>
→ 1 + 2 * <e>
→ 1 + 2 * 3



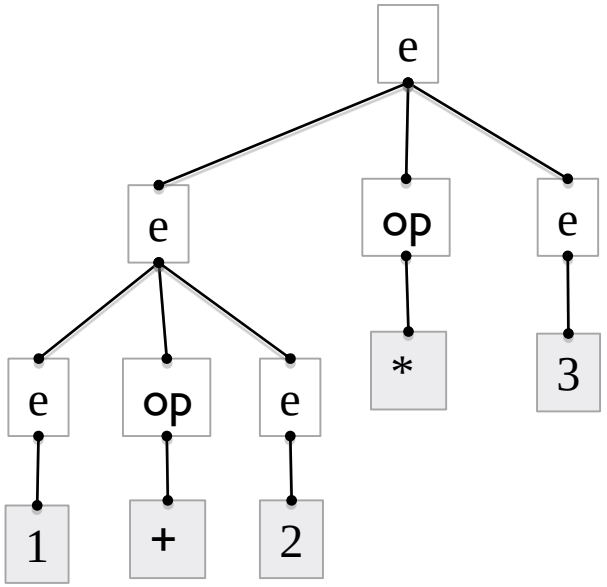
Two derivations for 1+2*3

<e> → <e> <op> <e>
<e> → 1|2|3
<op> → +|-|*|/

<e> → <e> <op> <e>
→ 1 <op> <e>
→ 1 + <e>
→ 1 + <e> <op> <e>
→ 1 + 2 <op> <e>
→ 1 + 2 * <e>
→ 1 + 2 * 3



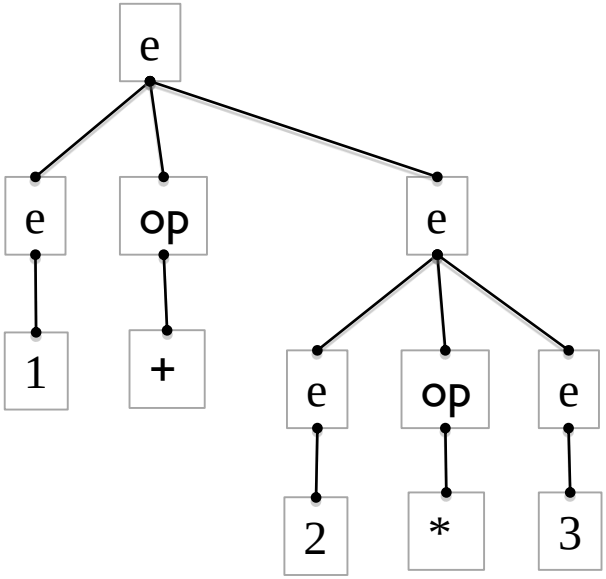
<e> → <e> <op> <e>
→ <e> <op> <e> <op> <e>
→ 1 <op> <e> <op> <e>
→ 1 + <e> <op> <e>
→ 1 + 2 <op> <e>
→ 1 + 2 * <e>
→ 1 + 2 * 3

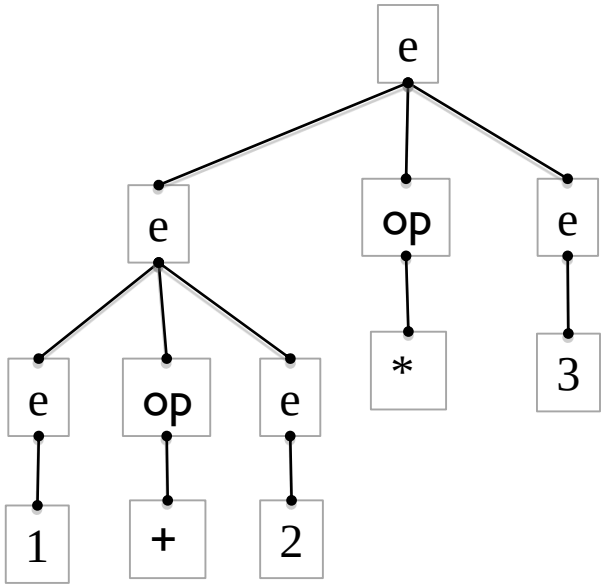


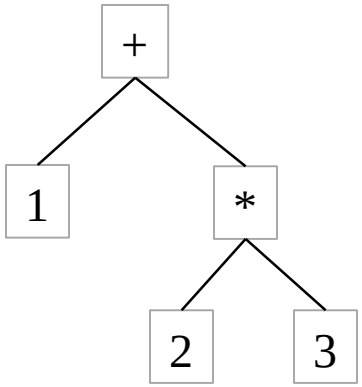
The leaves of the trees are terminals and correspond to the sentence

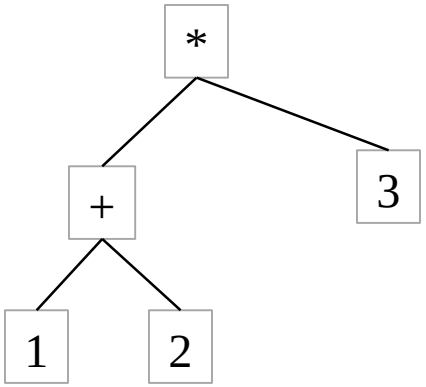
Two derivations for 1+2*3

<e> → <e> <op> <e>
<e> → 1|2|3
<op> → +|-|*|/







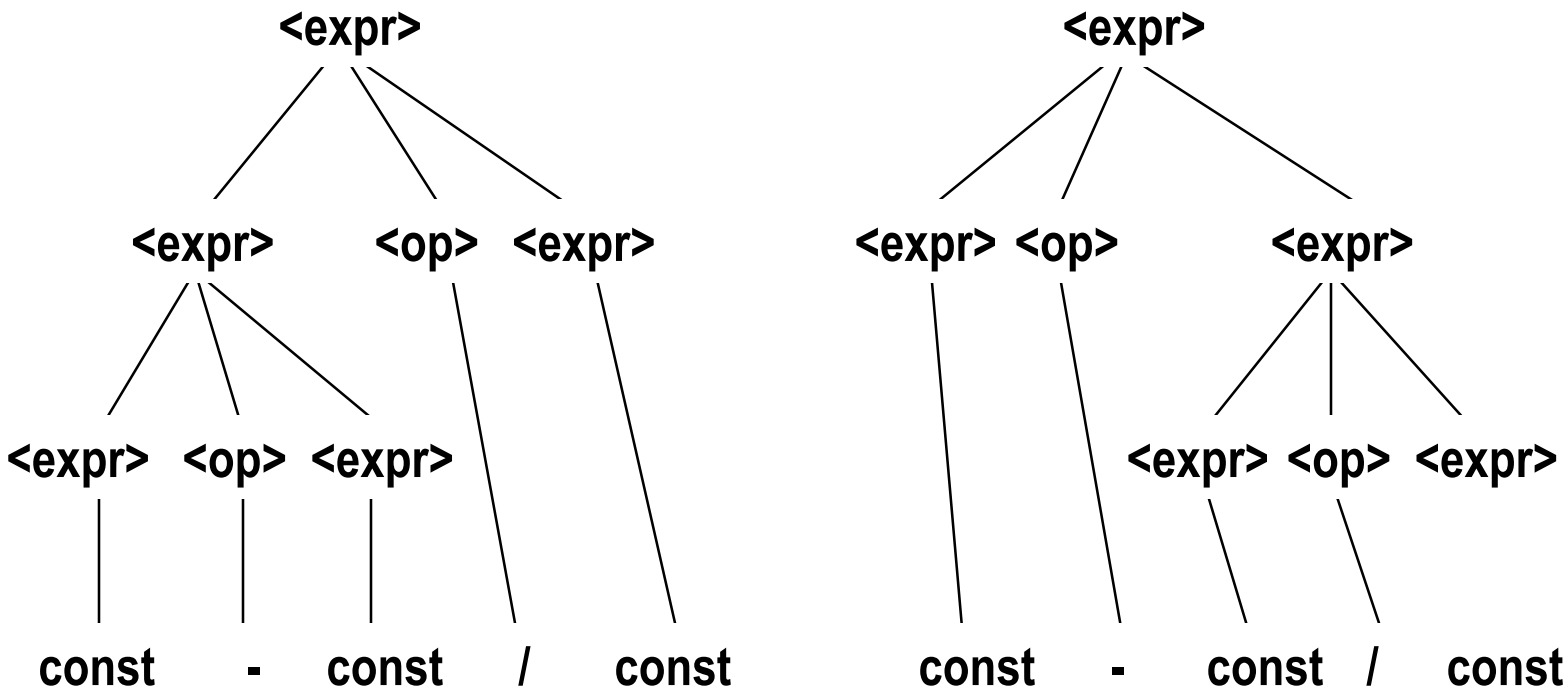


Lower trees represent the way we think about the sentence ‘meaning’

Example 2: An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

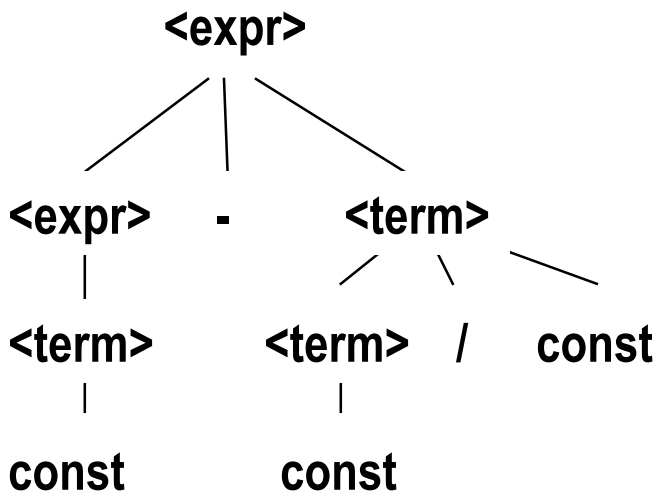
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

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



Summary

- The syntax of a programming language is usually defined using BNF or a context free grammar
- In addition to defining what programs are syntactically legal, a grammar also encodes meaningful or useful abstractions (e.g., block of statements)