Lecture 2

Describing Syntax and Semantics

Objectives

1. Differentiate Syntax from Semantics
2. The General Problem of Describing Syntax
3. Formal Methods of Describing Syntax
4. Attribute Grammars
5. Describing the Meanings of Programs: Dynamic Semantics

Syntax and Semantics

- Programming language syntax: how programs look, their form and structure
  - Syntax is defined using a kind of formal grammar
- Programming language semantics: what programs do, their behavior and meaning

An English Grammar

A sentence is a noun phrase, a verb, and a noun phrase.	<*S*> ::= <*NP*> <*V*> <*NP*>
A noun phrase is an article and a noun.	<*NP*> ::= <*A*> <*N*>
A verb is…	<*V*> ::= **`loves`** | **`hates`** | **`eats`**
An article is…	<*A*> ::= **`a`** | **`the`**
A noun is...	<*N*> ::= **`dog`** | **`cat`** | **`rat`**

start symbol

<*S*> ::= <*NP*> <*V*> <*NP*>

a production

<*NP*> ::= <*A*> <*N*>

<*V*> ::= `loves` | `hates` | `eats`

<*A*> ::= `a` | `the`

non-terminal symbols

<*N*> ::= `dog` | `cat` | `rat`

tokens

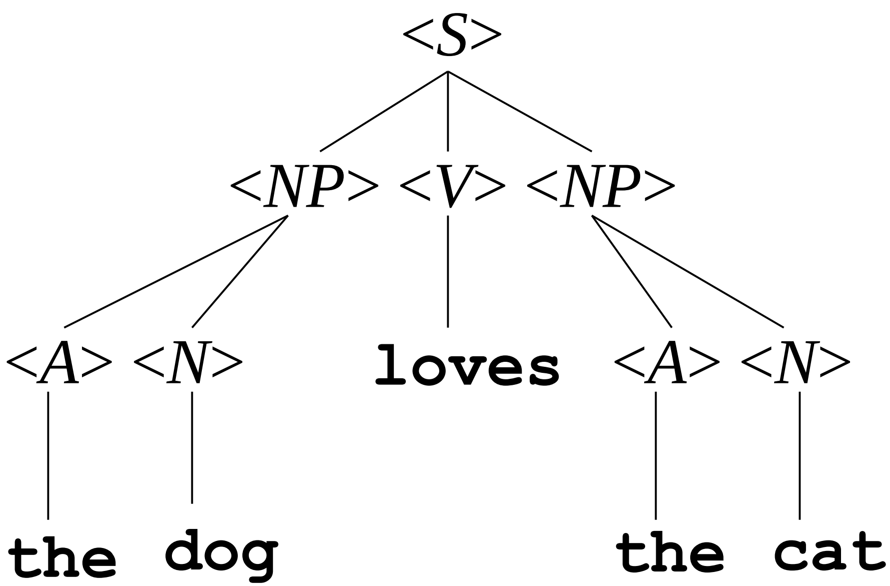
How The Grammar Works

- The grammar is a set of rules that say how to build a tree—a *parse tree*
- You put <*S*> at the root of the tree
- The grammar’s rules say how children can be added at any point in the tree
- For instance, the rule

  <*S*> ::= <*NP*> <*V*> <*NP*>

  says you can add nodes <*NP*>, <*V*>, and <*NP*>, in that order, as children of <*S*>

A Parse Tree



A Programming Language Grammar

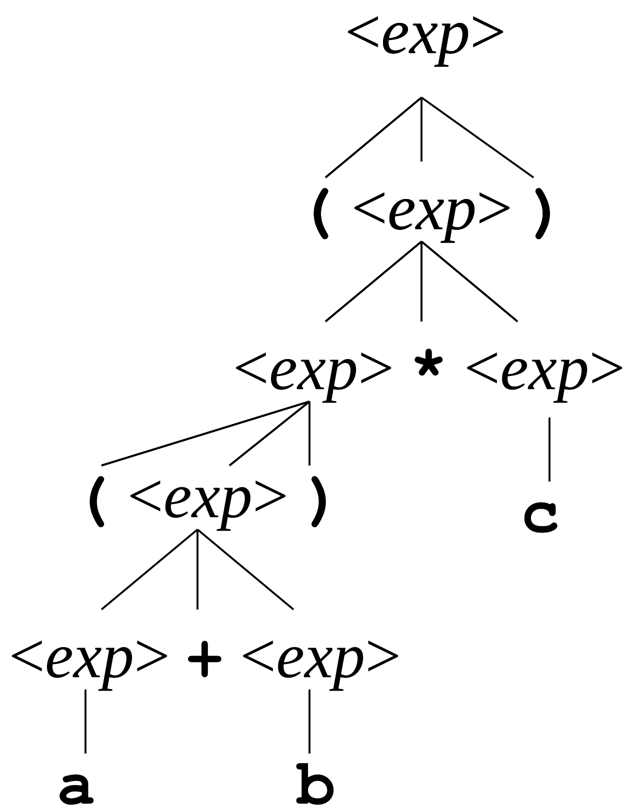
<exp> ::= *<exp>* `+` *<exp>* | *<exp>* `*` *<exp>* |

`(` *<exp>* `)` | `a` | `b` | `c`

- An expression can be the sum of two expressions, or the product of two expressions, or a parenthesized subexpression
- Or it can be one of the variables `a`, `b` or `c`

A Parse Tree

((a+b)*c)



Formal Definition

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum, begin`)
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
  - Detailed discussion next meeting
- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

- Backus-Naur Form and Context-Free Grammars
  - Most widely known method for describing programming language syntax
- Extended BNF
  - Improves readability and writability of BNF
- Grammars and Recognizers

BNF and Context-Free 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

Context-Free Grammar

A Context-Free Grammar G is a quadruple **(V , Σ, R, S)** where:

V : is an alphabet,

Σ : set of Terminals,

R :set of Rules

S : the start symbol which is an element of **(V - Σ)**.

Context-Free Grammar

Example:

$L = \{a^n b^n \, / \, n >= 0\}$

CFG $G = (V , \Sigma, R, S)$

$V = \{ S, a, b \}$

$\Sigma = \{ a, b \}$

$R = \{ \; S \rightarrow aSb,$

$\quad S \rightarrow e$

$\}$

Context-Free Grammar

a. $L = \{ w \in \{a, b\}^* \,/\, |w| \text{ is even } \}$

b. $L = \{ w \in \{a, b\}^* \,/\, w \text{ contains the substring } ab \}$

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars
  - BNF is a*metalanguage* 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*)

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules
  - Examples of BNF rules:

```
<ident_list> → identifier |
               identifer, <ident_list>

<if_stmt> → if <logic_expr> then <stmt>
```

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 a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
        | ident, <ident_list>
```

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

An Example Grammar

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

An example derivation

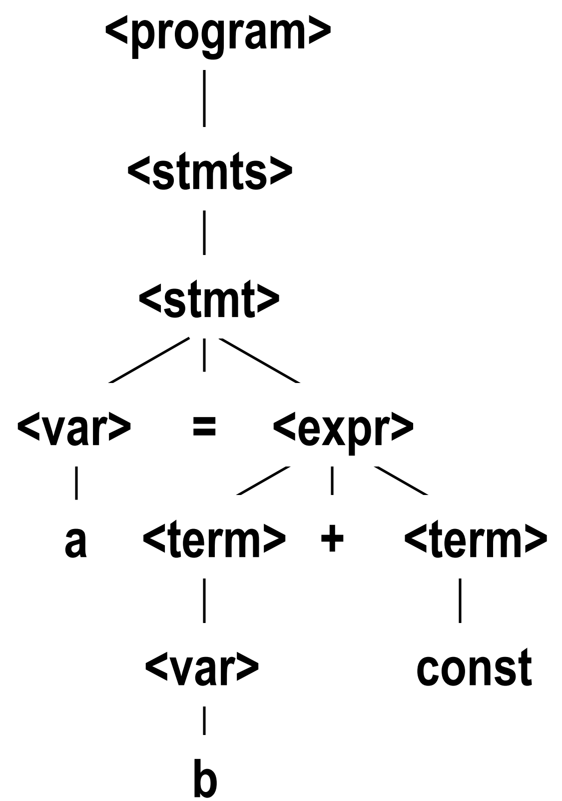
```
<program> => <stmts> => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

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
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



Constructing Grammars

- Most important trick: divide and conquer
- Example: the language of Java declarations: a type name, a list of variables separated by commas, and a semicolon
- Each variable can be followed by an initializer:

```
float a;
boolean a,b,c;
int a=1, b, c=1+2;
```

Example

```
<var-dec> → <type-name> <declarator-list> ;

<type-name> → boolean | byte | short | int
              | long | char | float | double

<declarator-list> → <declarator>
                    | <declarator> , <declarator-list>

<declarator> ::= <variable-name>
                 | <variable-name> = <expr>
```

Extended BNF

- Optional parts are placed in brackets ([ ])

```
<proc_call> -> ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces ({ })

```
<ident> → letter {letter|digit}
```

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
             | <expr> - <term>
                 | <term>
<term> → <term> * <factor>
             | <term> / <factor>
             | <factor>
```

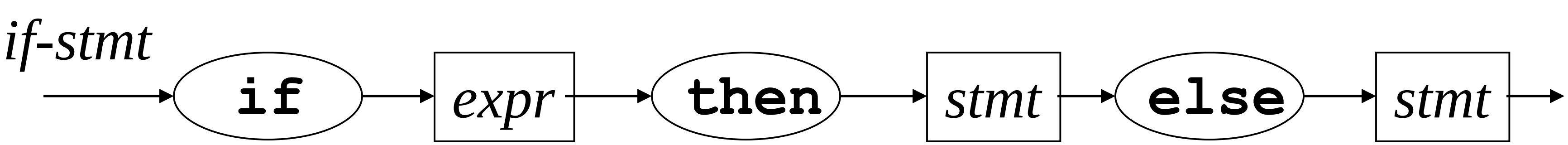
- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Syntax Diagrams

- Syntax diagrams ("railroad diagrams")
- Start with an EBNF grammar
- A simple production is just a chain of boxes (for nonterminals) and ovals (for terminals):

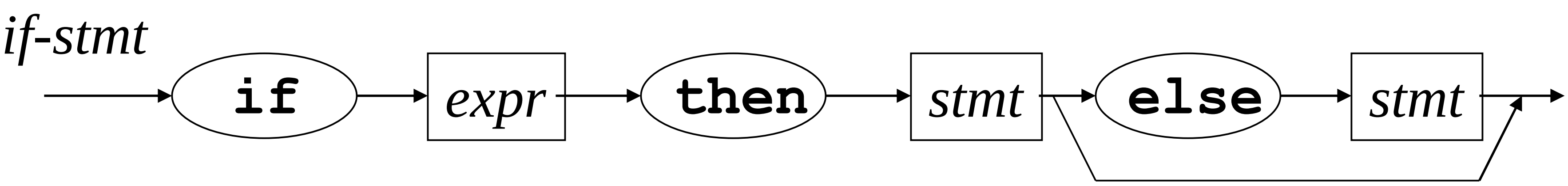
$$<\textit{if-stmt}> \rightarrow \textbf{\texttt{if}}\ <\textit{expr}>\ \textbf{\texttt{then}}\ <\textit{stmt}>\ \textbf{\texttt{else}}\ <\textit{stmt}>$$



Bypasses

- Square-bracket pieces from the EBNF get paths that bypass them

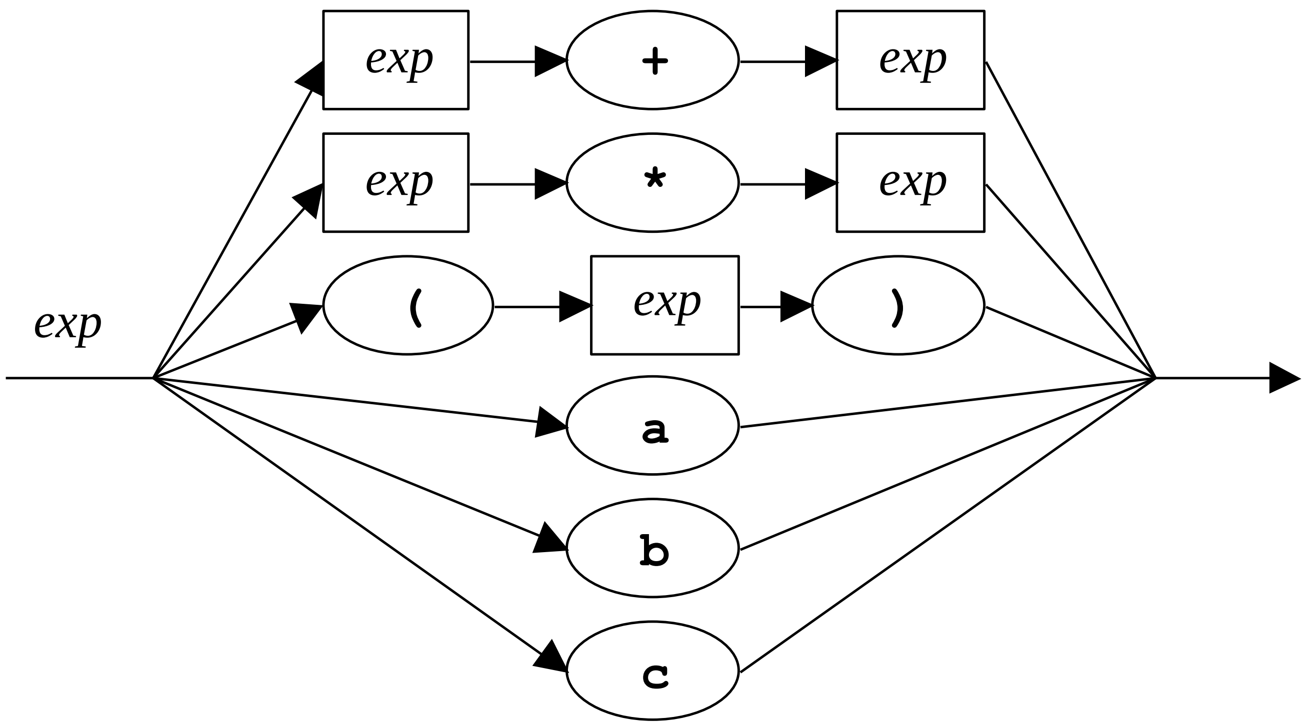
<if-stmt> → **`if`** *<expr>* **`then`** *<stmt>* [**`else`** *<stmt>*]



Branching

- Use branching for multiple productions

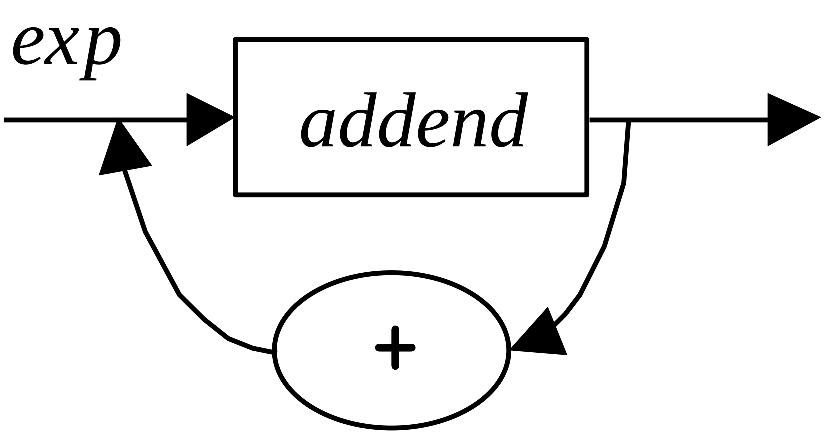
<exp> → *<exp>* **+** *<exp>* | *<exp>* ***** *<exp>* | **(** *<exp>* **)**
| **a** | **b** | **c**



Loops

- Use loops for EBNF curly brackets

$$<exp> \rightarrow <addend> \{ \textbf{+} <addend> \}$$



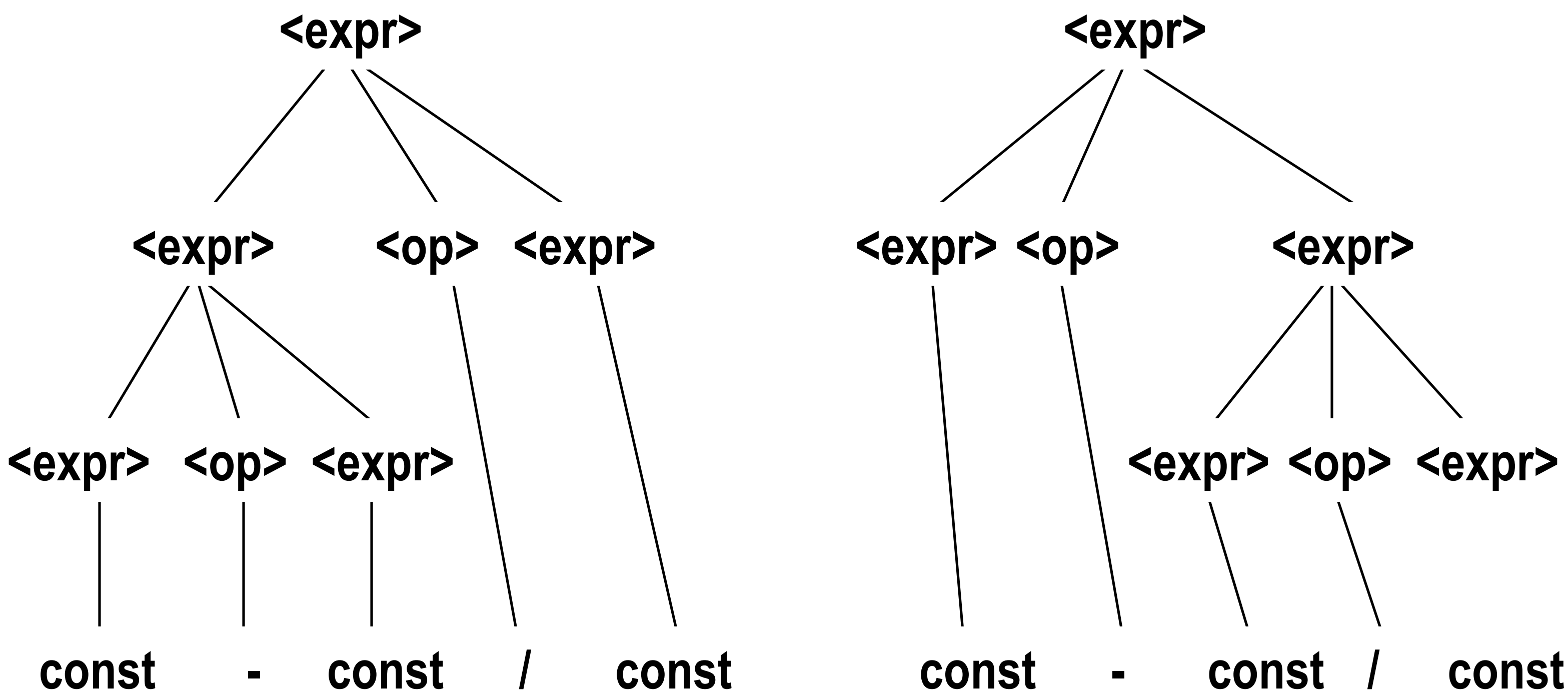
Ambiguity in Grammars

- A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

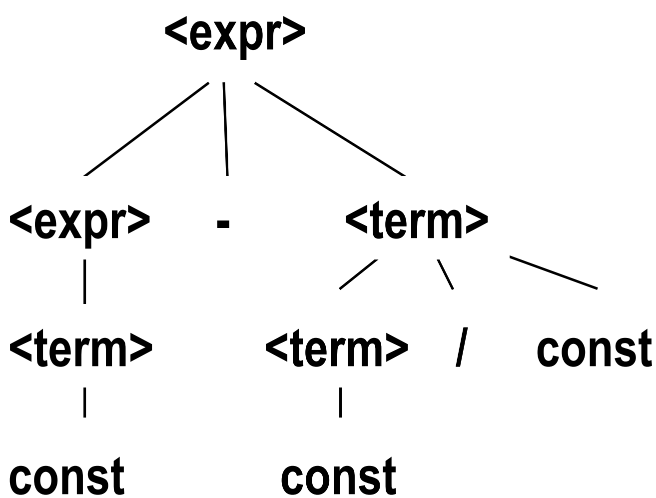
<op> → / | -



An Unambiguous Expression Grammar

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

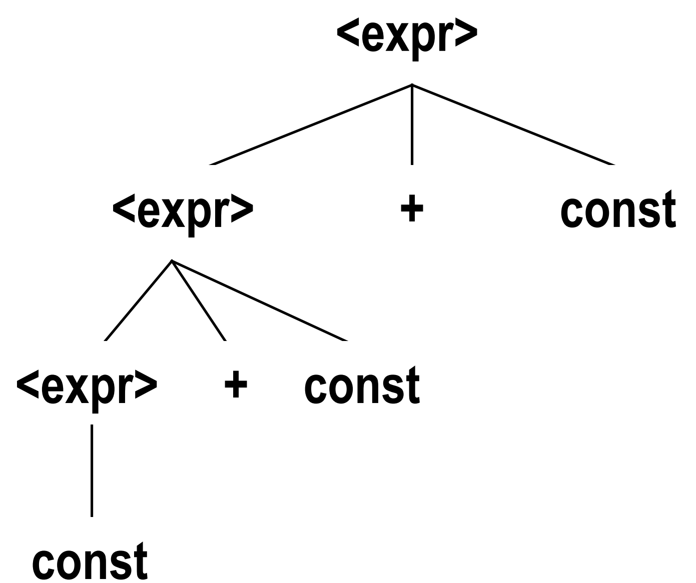
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const
  (ambiguous)
<expr> -> <expr> + const  |  const
  (unambiguous)
```



- Special syntax for frequently-used simple operations like addition, subtraction, multiplication and division
- The word *operator* refers both to the token used to specify the operation (like **+** and *****) and to the operation itself
- Usually predefined, but not always
- Usually a single token, but not always

Operator Terminology

- *Operands* are the inputs to an operator, like **`1`** and **`2`** in the expression **`1+2`**
- *Unary* operators take one operand: **`-1`**
- *Binary* operators take two: **`1+2`**
- *Ternary* operators take three: **`a?b:c`**

More Operator Terminology

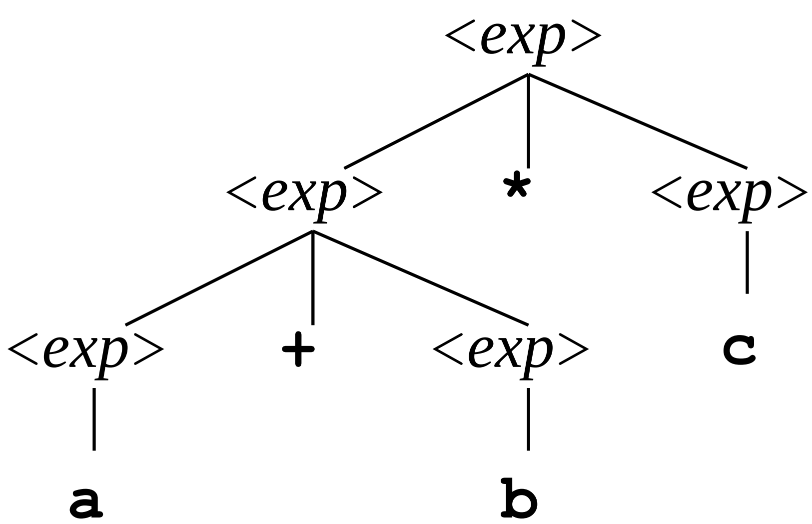
- In most programming languages, binary operators use an *infix* notation: **`a + b`**
- Sometimes you see *prefix* notation: **`+ a b`**
- Sometimes *postfix* notation: **`a b +`**
- Unary operators, similarly:
  - (Can't be infix, of course)
  - Can be prefix, as in **`-1`**
  - Can be postfix, as in **`a++`**

Working Grammar

G4**:** *<exp>* **::=** *<exp>* **+** *<exp>*
| *<exp>* ***** *<exp>*
| **(***<exp>***)**
| **a** | **b** | **c**

This generates a language of arithmetic expressions using parentheses, the operators **+** and *****, and the variables **a**, **b** and **c**

Issue #1: Precedence



Our grammar generates this tree for **`a+b*c`**. In this tree, the addition is performed before the multiplication, which is not the usual convention for operator *precedence*.

Operator Precedence

- Applies when the order of evaluation is not completely decided by parentheses
- Each operator has a *precedence level,* and those with higher precedence are performed before those with lower precedence, as if parenthesized
- Most languages put `*` at a higher precedence level than `+`, so that

    `a+b*c` = `a+(b*c)`

Precedence In The Grammar

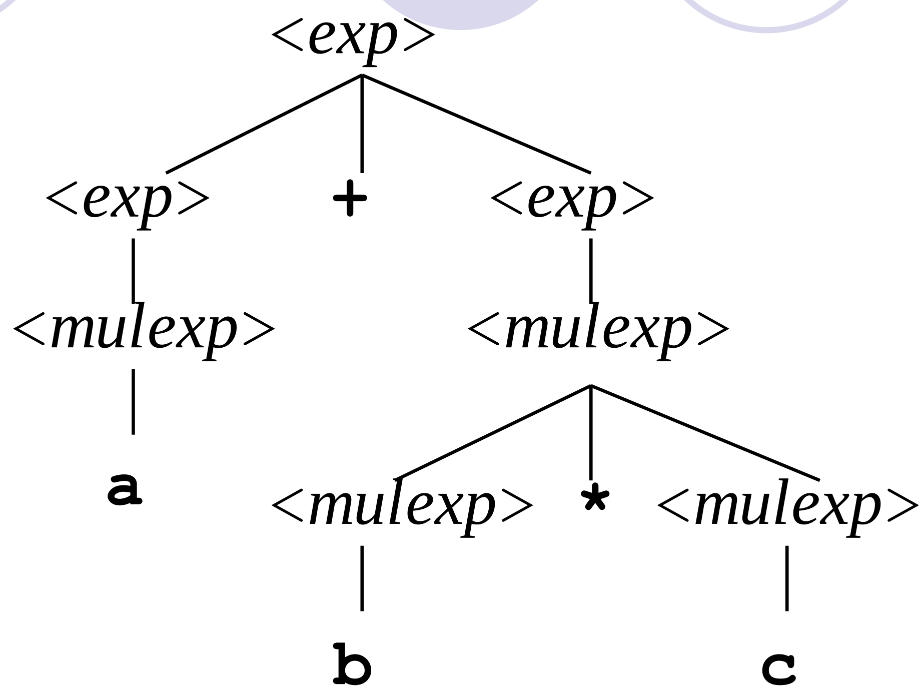
```
G4:  <exp> ::= <exp> + <exp>
             | <exp> * <exp>
             | (<exp>)
             | a | b | c
```

To fix the precedence problem, we modify the grammar so that it is forced to put `*` below `+` in the parse tree.

```
G5:  <exp> ::= <exp> + <exp> | <mulexp>
     <mulexp> ::= <mulexp> * <mulexp>
                | (<exp>)
                | a | b | c
```

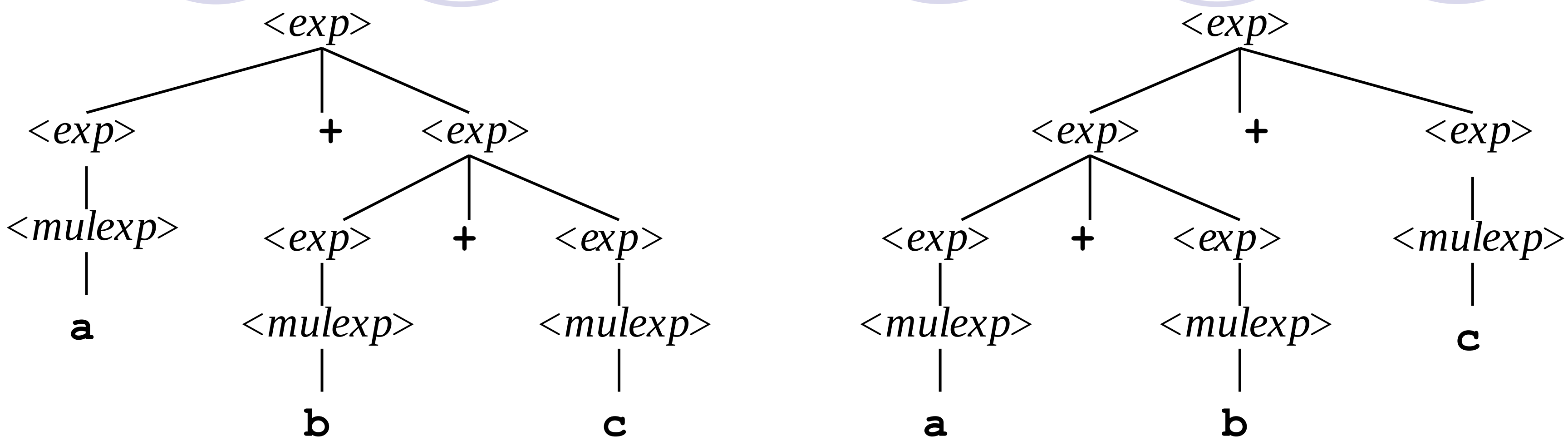
Correct Precedence

G5 parse tree:



Our new grammar generates this tree for **`a+b*c`**. It generates the same language as before, but no longer generates parse trees with incorrect precedence.

Issue #2: Associativity



Our grammar G5 generates both these trees for **`a+b+c`**. The first one is not the usual convention for operator *associativity*.

Operator Associativity

- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative* operators group left to right: `a+b+c+d = ((a+b)+c)+d`
- *Right-associative* operators group right to left: `a+b+c+d = a+(b+(c+d))`
- Most operators in most languages are left-associative, but there are exceptions

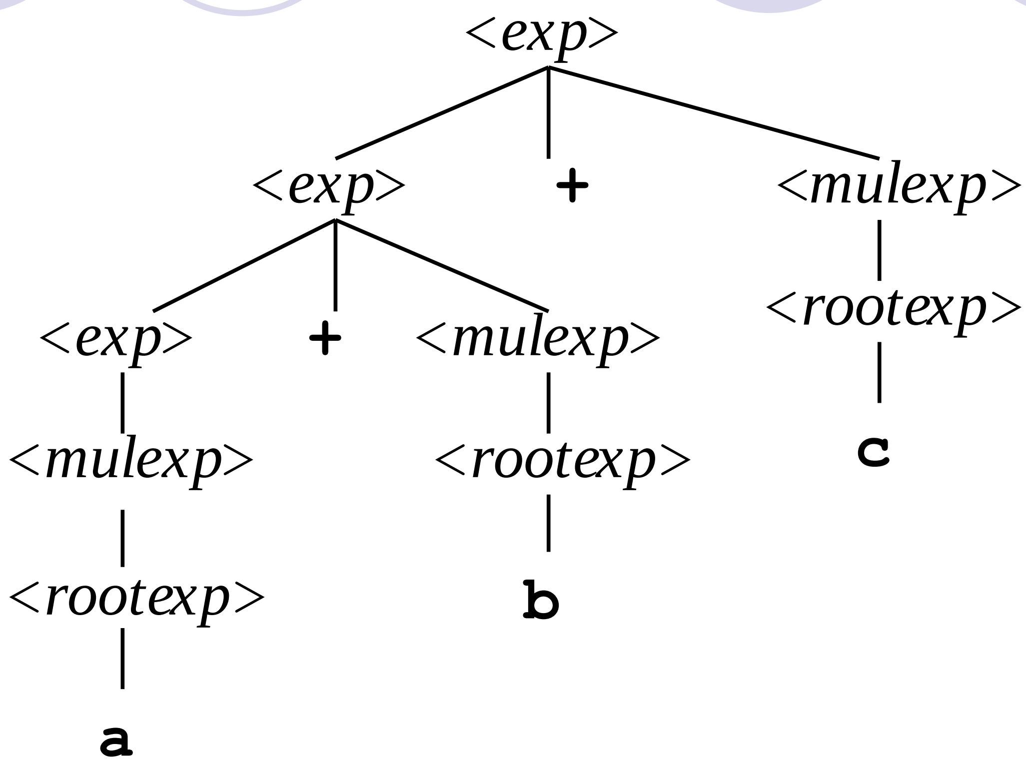
Associativity In The Grammar

```
G5:   <exp> ::= <exp> + <exp> | <mulexp>
      <mulexp> ::= <mulexp> * <mulexp>
                 | (<exp>)
                 | a | b | c
```

To fix the associativity problem, we modify the grammar to make trees of **+**s grow down to the left (and likewise for *****s)

```
G6:   <exp> ::= <exp> + <mulexp> | <mulexp>
      <mulexp> ::= <mulexp> * <rootexp> | <rootexp>
      <rootexp> ::= (<exp>)
                  | a | b | c
```

Correct Associativity



Our new grammar generates this tree for **`a+b+c`**. It generates the same language as before, but no longer generates trees with incorrect associativity.

Attribute Grammars

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Additions to CFGs to carry some semantic info along parse trees
- Primary value of attribute grammars (AGs):
  - Static semantics specification
  - Compiler design (static semantics checking)

Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

Semantics

- Axiomatic Semantics
  - Based on formal logic (predicate calculus)
  - Original purpose: formal program verification
  - Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
  - The expressions are called assertions

Semantics

- Denotational Semantics
  - Based on recursive function theory
  - The most abstract semantics description method
  - Originally developed by Scott and Strachey (1970)

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational