Principles of Programming Languages

Syntax and Semantic

http://fm.zju.edu.cn

Introduction

How to Say It Right?

- Suppose you mean to say

  浙江大学位于杭州市。

- What is wrong with this "sentence"?

  Zhejiang University in Hangzhou.

- Can it convey the meaning?
  - Wrong grammar often obscure the meanings
- What about these "sentences"?

  Zhejiang University sits in Hangzhou.

  Hangzhou is in Zhejiang University.

Description of a Language

- Syntax: 语法the form or structure of the expressions, statements, and program units
- Semantics: 语义the meaning of the expressions, statements, and program units
    - What programs do, their behavior and meaning
- So, when we say one's English grammar is wrong, we actually mean _________ error?

What Kind of Errors They Have?

Zhejiang University in Hangzhou.

Zhejiang University am in Hangzhou.

Zhejiang University sits in Hangzhou.

Hangzhou is in Zhejiang University.

Describing Syntax and Semantics

- Syntax is defined using some kind of rules
    - Specifying how statements, declarations, and other language constructs are written
- Semantics is more complex and involved. It is harder to define, e.g., natural language doc.
- Example: **if** statement
    - Syntax: **if (<expr>) <statement>**
    - Semantics: if **<expr>** is true, execute **<statement>**
- Detecting syntax error is easier, semantics error is much harder

General Problem of Describing Syntax

What is a Language?

- In programming language terminologies, a language is a set of sentences
- A sentence is a string of characters over some alphabet
  - The meaning of a "sentence" is very general. In English, it may be an English sentence, a paragraph, or all the text in a book, or hundreds of books, …
- Every C program, if can be compiled properly, is a sentence of the C language
  - No matter whether it is "hello world" or a program with several million lines of code

A Sentence in C Language

- The "Hello World" program is a sentence in C

```
main()
{  printf("hello, world!\n"); }
```

- What about its alphabet?
  - For illustration purpose, let us define the alphabet as
    a → identifier   b → string   c→'('
    d→')'   e→'{'   f→'}'   g→';'
  - So, symbolically "Hello World" program can be represented by the sentence: acdeacbdgf
    where "main" and "printf" are identifiers and "hello, world!\n" is a string

Sentence and Language

- So, we say that <u>acdeacbdgf</u> is a sentence of (or, in) the C language, because it represents a legal program in C
  - Note: "legal" means syntactically correct
- How about the sentence <u>acdeacbdf</u>?
  - It represents the following program:

```
main()
{  printf("hello, world!\n") }
```

  - Compiler will say there is a syntax error
  - In essence, it says the sentence <u>acdeacbdf</u> is not in C language

So, What a C Compiler Does?

- Frontend: check whether the program is "a sentence of the C language"
  - Lexical analysis: translate C code into corresponding "sentence" (intermediate representation, IR) → Ch. 4
  - Syntax analysis: check whether the sentence is a sentence in C → Ch. 4
    - Not much about what it means → semantics
- Backend: translate from "sentence" (IR) into object code
  - Local and global optimization
  - Code generation: register and storage allocation, …

Definition of a Language

- The syntax of a language can be defined by a set of syntax rules
- The syntax rules of a language specify which sentences are in the language, i.e., which sentences are legal sentences of the language
- So when we say

你的句子不合英文文法。

we actually say

你的句子不在英文裡。

Syntax Rules

- Consider a special language X containing sentences such as

ZU is in Hangzhou.

ZU belongs to Hangzhou.

- A general rule of the sentences in X may be

A sentence consists of a noun followed by a verb, followed by a preposition, and followed by a noun,

where a noun is a place

a verb can be "is" or "belongs" and

a preposition can be "in" or "to"

Syntax Rules

hierarchical structure of language

- A more concise representation:

  <sentence> → <noun> <verb> <preposition> <noun>

  <noun> → place

  <verb> → "is" | "belongs"

  <preposition> → "in" | "to"

- With these rules, we can generate followings:

  ZU is in Hangzhou

  Hangzhou is in ZU

  Hangzhou belongs to ZU

- They are all in language X
  - Its alphabet includes "is", "belongs", "in", "to", place

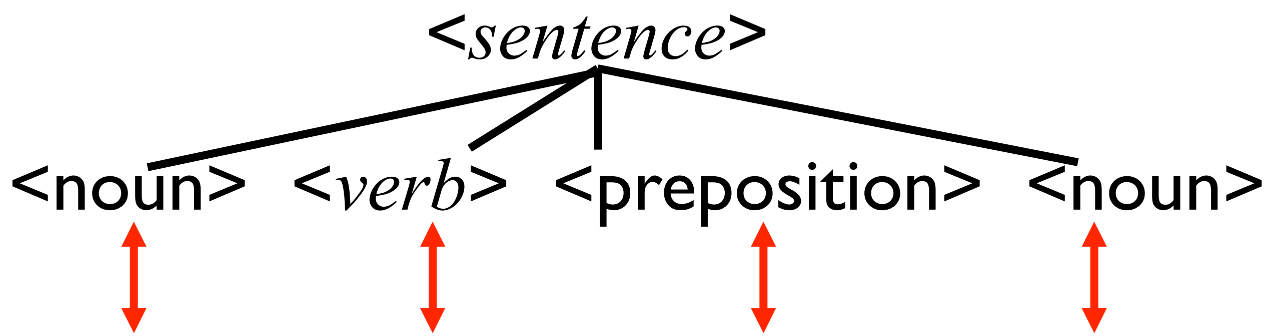
Checking Syntax of a Sentence

- How to check if the following sentence is in the language X?

  ZU belongs in Hangzhou

- Idea: check if you can generate that sentence
  → This is called parsing
- How?
  Try to match the input sentence with the structure of the language

Matching the Language Structure



So, the sentence is in the language X!

ZU belongs in Hangzhou

The above structure is called a parse tree

Summary: Language, Sentence

	Language	English	Chinese	C
Syntax rules	$\in$			
	Sentence	How are you? ZU is in Hangzhou.		
	Alphabet	a,b,c,d,...		

Formal Methods of Describing Syntax

Formal Description of Syntax

Most widely known methods for describing syntax:

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Define a class of languages: context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58
  - Equivalent to context-free grammars

BNF Terminologies

- A lexeme（语素） is the lowest level syntactic unit of a language (e.g., ZU, Hangzhou, is, in)
- A token（标记） is a category of lexemes (e.g., place)
- A BNF grammar consists of four parts:
  - The set of tokens and lexemes (terminals（终结符）)
  - The set of non-terminals（非终结符）, e.g., <sentence>, <verb>
  - The start symbol（起始符号）, e.g., <sentence>
  - The set of production rules, e.g.,

<sentence> → <noun> <verb> <preposition> <noun>

<noun> → place

<verb> → “is” | “belongs” <preposition> → “in” | “to”

BNF Terminologies

- Tokens and lexemes are smallest units of syntax
  - Lexemes appear literally in program text
- Non-terminals stand for larger pieces of syntax
  - Do NOT occur literally in program text
  - The grammar says how they can be expanded into strings of tokens or lexemes
- The start symbol is the particular non-terminal that forms the starting point of generating a sentence of the language

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS)
  - LHS is a single non-terminal → context-free
  - RHS contains one or more terminals or non-terminals
  - A rule tells how LHS can be replaced by RHS, or how RHS is grouped together to form a larger syntactic unit (LHS) → traversing the parse tree up and down
  - A nonterminal can have more than one RHS
  - A syntactic list can be described using recursion

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

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

`<program>` is the start symbol

`a`, `b`, `c`, `const`, `+`, `-`, `;`, `=` are the terminals

Derivation

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

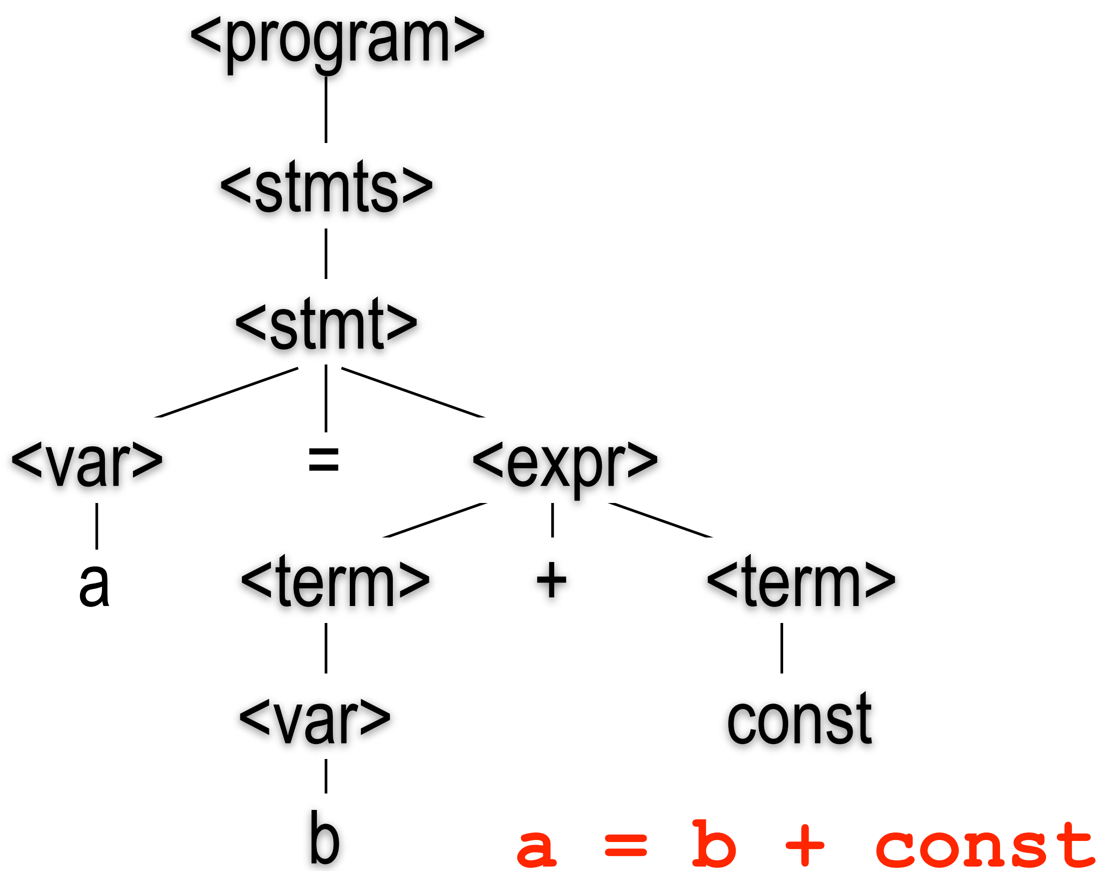
```
<program> => <stmts>
   => <stmt>
            => <var> = <expr>
   => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Derivation（推导）

- Every string of symbols in the derivation is a sentential form（句型）
- 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



Grammar and Parse Tree

- The grammar can be viewed as a set of rules that say how to build a parse tree
- You put <S> at the root of the tree
- Add children to every non-terminal, following any one of the rules for that non-terminal
- Done when all the leaves are tokens
- Read off leaves from left to right—that is the string derived by the tree
  - e.g., in the case of C language, the leaves form the C program, despite it has millions of lines of code

How to Check a Sentence?

- What we have discussed so far are how to generate/derive a sentence
- For compiler, we want the opposite
→ check whether the input program (or its corresponding sentence) is in the language!
- How to do?
  - Use tokens in the input sentence one by one to guide which rules to use in derivation or to guide a reverse derivation

Compiler Note

- Compiler tries to build a parse tree for every program you want to compile, using the grammar of the programming language
- Given a CFG (Context-Free Grammar), a recognizer for the language generated by the grammar can be algorithmically constructed, e.g., yacc
  - The compiler course discusses algorithms for doing this efficiently

Issues in Grammar Definitions: Ambiguity, Precedence, Associativity, …

Three "Equivalent" Grammars

G1: *<subexp>* → **a** | **b** | **c** | *<subexp>* **-** *<subexp>*

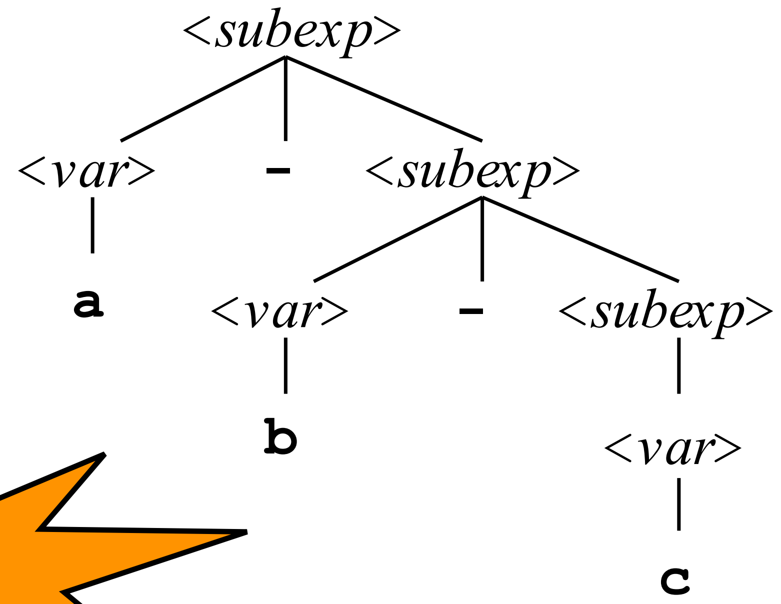
G2: *<subexp>* → *<var>* **-** *<subexp>* | *<var>*
<var> → **a** | **b** | **c**

G3: *<subexp>* → *<subexp>* - *<var>* | *<var>*
<var> → **a** | **b** | **c**

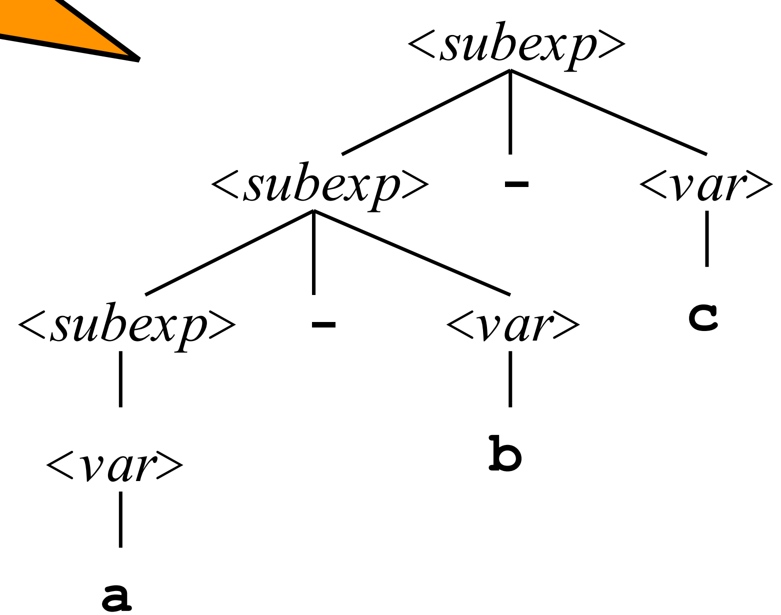
These grammars all define the same language: the language of strings that contain one or more **a**s, **b**s or **c**s separated by minus signs, e.g., **`a-b-c`**. But...

What are the differences?

G2 parse tree:



G3 parse tree:



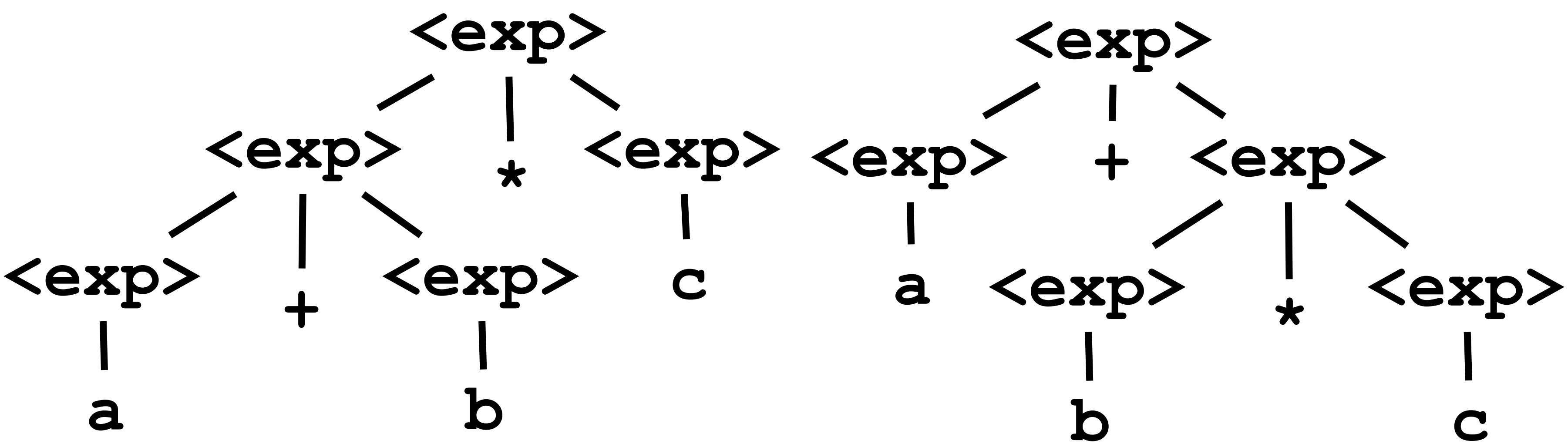
Ambiguity in Grammars

- If a sentential form can be generated by two or more distinct parse trees, the grammar is said to be ambiguous, because it has two or more different meanings
- Problem with ambiguity:
  - Consider the following grammar and the sentence **a +b*c**

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

An Ambiguous Grammar

- Two different parse trees for **a+b*c**



Means (a+b)*c

Means a+(b*c)

Consequences

- The compiler will generate different codes, depending on which parse tree it builds
  - According to convention, we would like to use the parse tree at the right, i.e., performing a+(b*c)
- Cause of the problem:
  Grammar lacks semantic of 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

Putting Semantics into Grammar

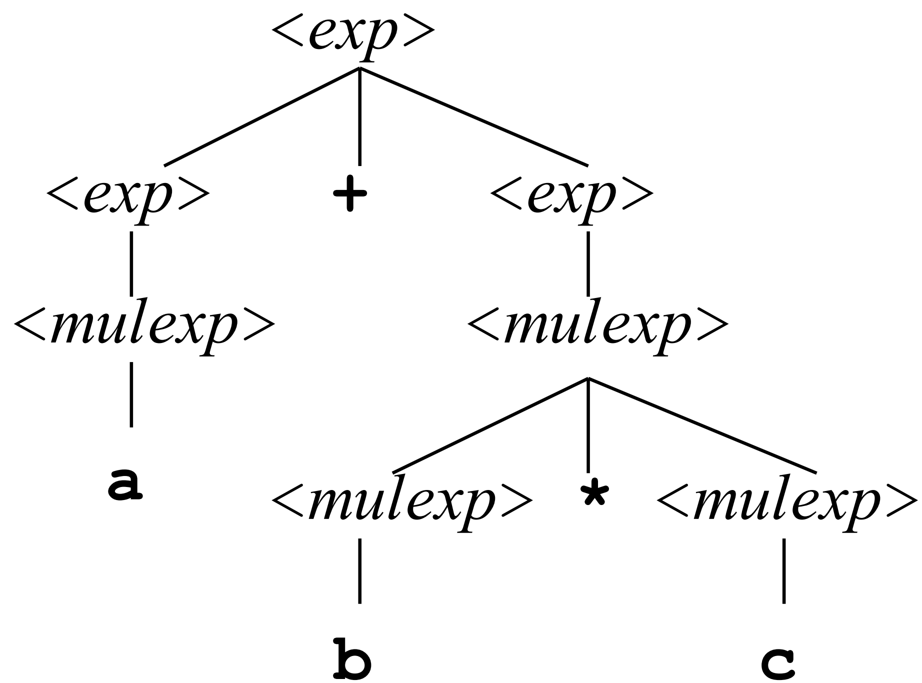
```
<exp> → <exp> + <exp> | <exp> * <exp>
      | (<exp>) | a | b | c
```

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

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

Note the hierarchical structure of the production rules

Correct Precedence

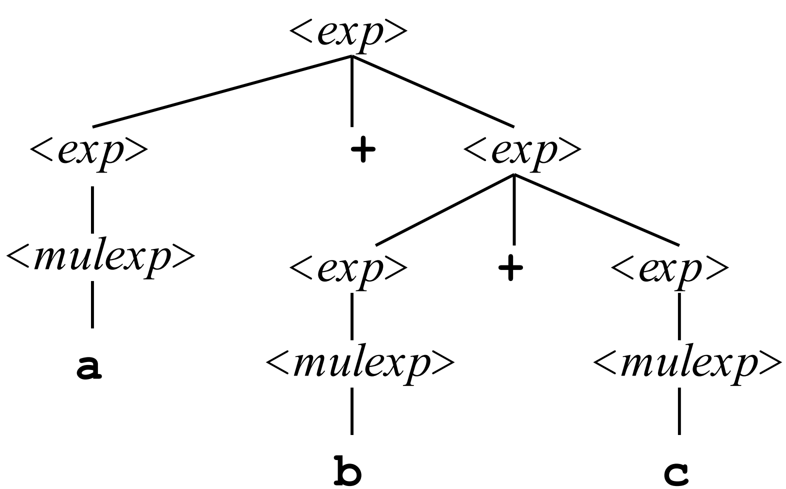


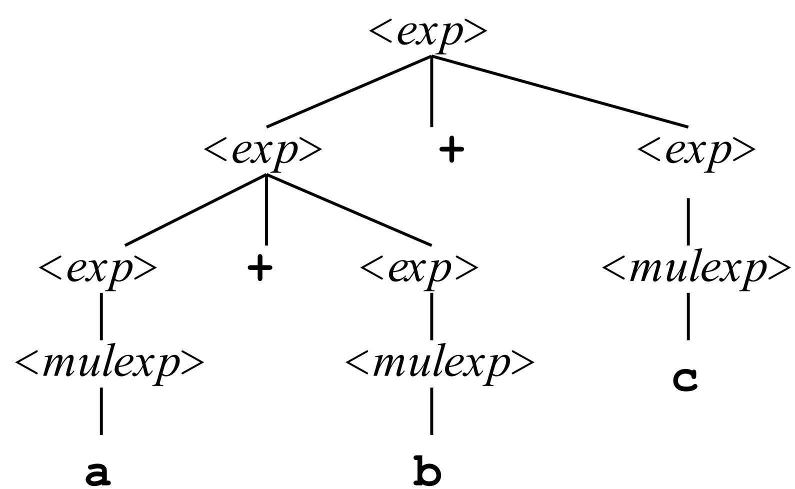
Our new grammar generates same language as before, but no longer generates parse trees with incorrect precedence.

Semantics of Associativity

- Grammar can also handle the semantics of operator associativity（结合律）

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





Operator Associativity

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

`a<<b<<c` — most operators are left-associative

`a=b=0` — right-associative (assignment)

Associativity Matters

- Addition is associative in mathematics?

$$(A + B) + C = A + (B + C)$$

- Addition is associative in computers?

- Subtraction and divisions are associative in mathematics?
- Subtraction and divisions are associative in computers?

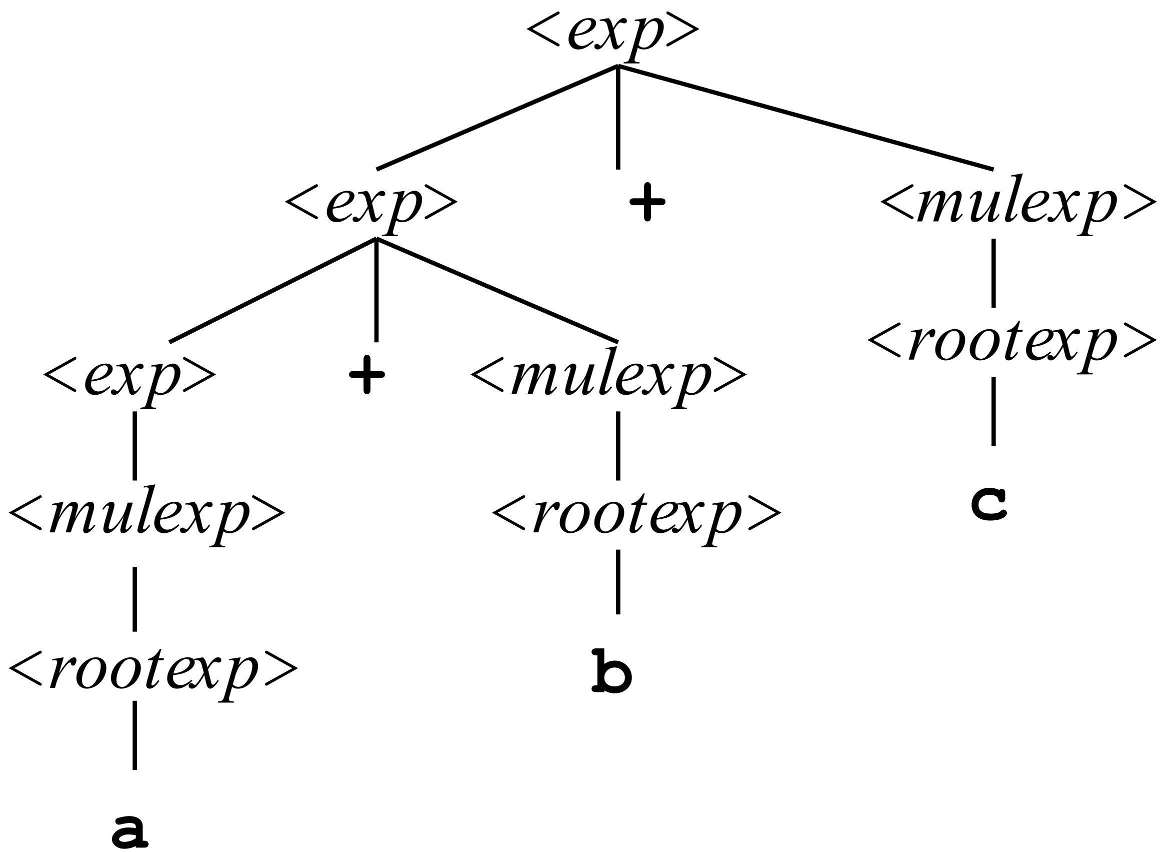
Associativity in the Grammar

```
<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)

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

Correct Associativity



Dangling Else in Grammars

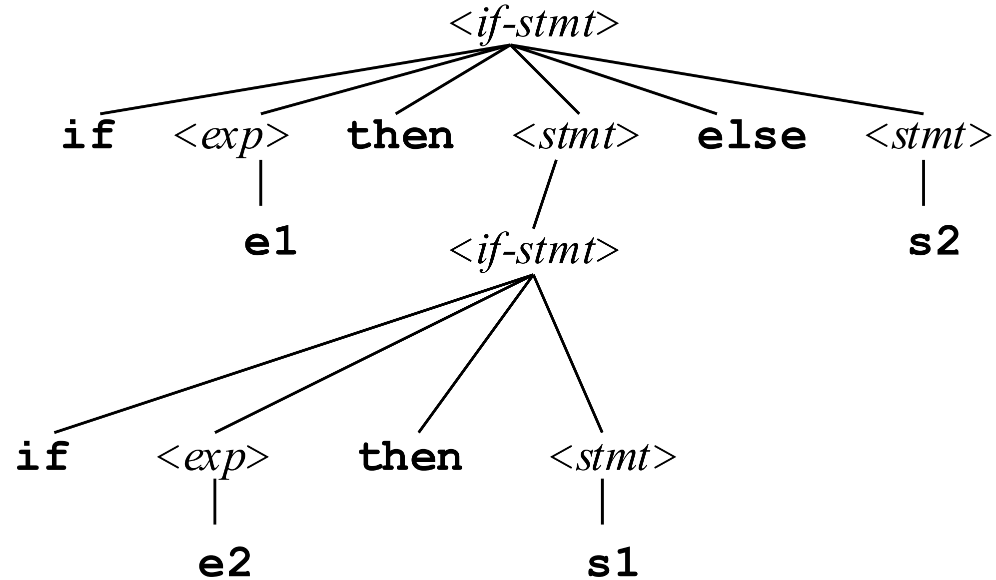
```
<stmt> → <if-stmt> | s1 | s2
<if-stmt> → if <expr> then <stmt> else
<stmt>
              | if <expr> then <stmt>
<expr> → e1 | e2
```

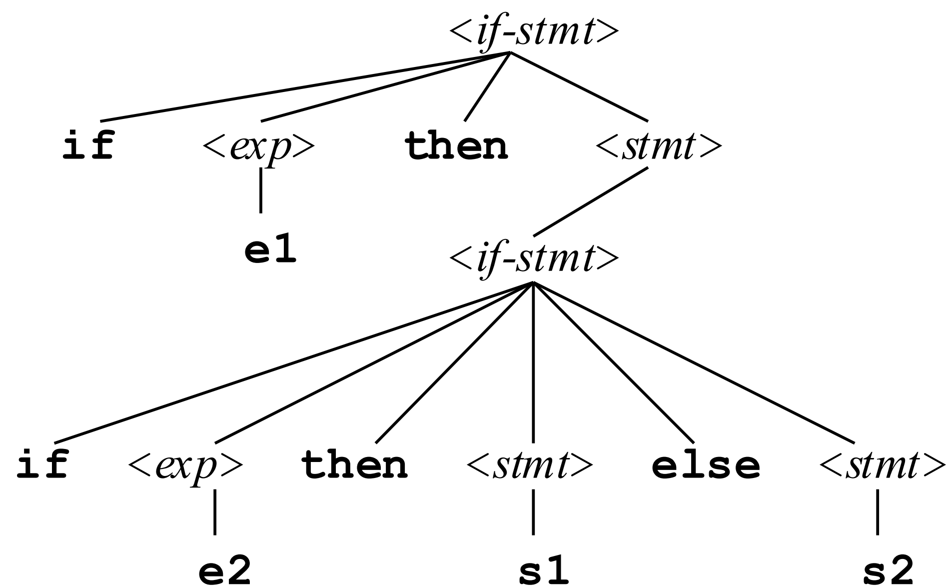
- This grammar has a classic "dangling-else ambiguity." Consider the statement

```
if e1 then if e2 then s1 else s2
```

Different Parse Trees

Most languages that have this problem choose this parse tree: **`else`** goes with nearest unmatched **`then`**





Eliminating the Ambiguity

<*stmt*> → <*if-stmt*> | s1 | s2

<*if-stmt*> → if <*expr*> then <*stmt*> else <*stmt*>

| if <*expr*> then <*stmt*>

<*expr*> → e1 | e2

If this expands into an **`if`**, that **`if`** must already have its own **`else`**.

First, we make a new non-terminal <*full-stmt*> that generates everything <*stmt*> generates, except that it can not generate **`if`** statements with no **`else`**:

<*full-stmt*> → <*full-if*> | s1 | s2

<*full-if*> → if <*expr*> then <*full-stmt*> else <*full-stmt*>

Eliminating the Ambiguity

```
<stmt> → <if-stmt> | s1 | s2
<if-stmt> → if <expr> then <full-stmt> else <stmt>
          | if <expr> then <stmt>
<expr> → e1 | e2
```

Then we use the new non-terminal here.

The effect is that the new grammar can match an **`else`** part with an **`if`** part only if all the nearer **`if`** parts are already matched.

Languages That Don't Dangle

- Some languages define if-then-else in a way that forces the programmer to be more clear
- ALGOL does not allow the `then` part to be another `if` statement, though it can be a block containing an `if` statement
- Ada requires each `if` statement to be terminated with an `end if`

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>}
```