

Principles of Programming Languages

Syntax and Semantic

Weng Kai
2019 Fall

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

$a \rightarrow \text{identifier}$ $b \rightarrow \text{string}$ $c \rightarrow '('$
 $d \rightarrow ')'$ $e \rightarrow \{'$ $f \rightarrow \}'$ $g \rightarrow ';'$

- 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 acdeacbdgf 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 acdeacbdf?

- It represents the following program:

```
main()
```

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

- Compiler will say there is a syntax error

- In essence, it says the sentence acdeacbdf 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

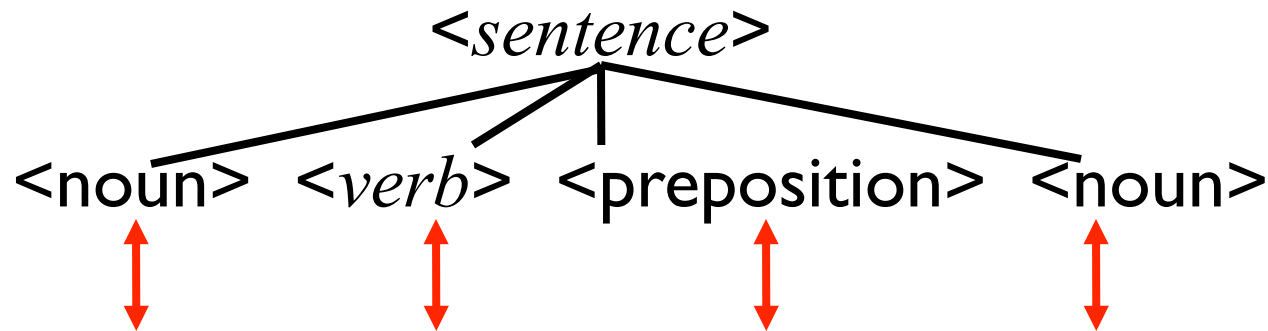
- ◆ Idea: check if you can generate that sentence

→ 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

\cup

Sentence

How are you?

ZU is in Hangzhou.

Alphabet

a,b,c,d,...

Formal Methods of Describing Syntax

What is BNF?

- BNF stands for “Backus-Naur Form,” after the people who invented it
- BNF is a *metalanguage*--a language used to describe another language
- BNF is a precise way of describing the syntax of a computer language
- BNF is really, really easy!

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

Who was John Backus?



- Backus invented FORTRAN (“FORMula TRANslator”), the first high-level language ever, circa 1954
- Major influence on the invention of functional programming in 1970’s
- Won the 1977 Turing Award for BNF and FORTRAN

Who was Peter Naur?



- Danish astronomer turned computer scientist
- Born in 1928; picture on left is from 1968

A Bit More History...

- BNF originally stood for “Backus Normal Form”
- In 1964, Donald Knuth wrote a letter published in Communications of the ACM in which he suggests it stand for Backus-Naur form instead
- This was for two reasons:
 - To recognize Naur’s contribution
 - BNF is not technically a “normal form”; this would imply that there would be only one correct way of writing a grammar

The syntax of BNF

- Any word or words enclosed in angle brackets, $\langle \rangle$, is a *nonterminal* -- it stands for some string (it's like a string variable)
- The symbol $::=$ or \rightarrow means “is defined as”
- The symbol $|$ means “or”
- Anything else is a *terminal* -- it stands for itself
- That's all!

What does BNF look like?

- Like this:

```
<number> ::= <digit> | <number> <digit>
```

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- “::=” means “is defined as” (some variants use “:=” instead)
- “|” means “or”
- Angle brackets mean a nonterminal
- Symbols without angle brackets are terminals

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>`

Examples

- $\langle \text{vowel} \rangle ::= a \mid e \mid i \mid o \mid u$
- $\langle \text{consonant} \rangle ::= b \mid c \mid d \mid f \mid g \mid h \mid j \mid k \mid l \mid m \mid n \mid p \mid q \mid r \mid s \mid t \mid v \mid w \mid x \mid y \mid z$
- $\langle \text{letter} \rangle ::= \langle \text{vowel} \rangle \mid \langle \text{consonant} \rangle$
- $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- $\langle \text{hex digit} \rangle ::= \langle \text{digit} \rangle \mid a \mid b \mid c \mid d \mid e \mid f$

Recursion

- Suppose that an “unsigned integer” consists of one or more digits (but there is no limit to the number of digits)
- $\langle \text{unsigned integer} \rangle ::=$
 $\langle \text{digit} \rangle$
 | $\langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$
- BNF makes heavy use of recursion

More examples

- Suppose that (in some computer language) a variable consists of letters, digits, and underscores, but must begin with a letter
- $\langle \text{variable} \rangle ::=$
 - $\langle \text{letter} \rangle$
 - $| \langle \text{variable} \rangle \langle \text{letter} \rangle$
 - $| \langle \text{variable} \rangle \langle \text{digit} \rangle$
 - $| \langle \text{variable} \rangle _$

BNF Examples

- `<while loop> ::= while (<condition>)
 <statement>`
- `<assignment statement> ::= <variable> =
 <expression>`
- `<statement list> ::= <statement> |
 <statement list> <statement>`
- `<unsigned integer> ::= <digit> |
 <unsigned integer><digit>`

An Example Grammar

$\langle \text{program} \rangle \rightarrow \langle \text{stmts} \rangle$

$\langle \text{stmts} \rangle \rightarrow \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmts} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{var} \rangle \rightarrow a \mid b \mid c \mid d$

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle - \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{var} \rangle \mid \text{const}$

$\langle \text{program} \rangle$ is the start symbol

$a, b, c, \text{const}, +, -, ;, =$ are the terminals

Sample Grammars

- Grammar for subset of English

Sentence \rightarrow **Noun Verb**

Noun \rightarrow Jack | Jill

Verb \rightarrow eats | bites

- Grammar for a digit

Digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Grammar for signed integers

SignedInteger \rightarrow **Sign Integer**

Sign \rightarrow + | -

Integer \rightarrow **Digit** | **Digit Integer**

- Grammar for subset of Java

Assignment \rightarrow **Variable = Expression**

Expression \rightarrow **Variable** | **Variable + Variable** | **Variable – Variable**

Length limitations

- Suppose that a variable is as before, but can only be six characters long
- $\langle \text{variable} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{letter} \rangle \langle \text{v2} \rangle$
 $\mid \langle \text{letter} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \mid \langle \text{letter} \rangle \langle \text{v2} \rangle$
 $\langle \text{v2} \rangle \langle \text{v2} \rangle$
 $\mid \langle \text{letter} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle$
 $\mid \langle \text{letter} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle \langle \text{v2} \rangle$
- $\langle \text{v2} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid _$
- BNF isn't great for setting limits on lengths!

Recursive structures

- A while loop can contain any number of other statements, including other while loops
- $\langle \text{while loop} \rangle ::= \text{while} (\langle \text{condition} \rangle) \{ \langle \text{statements} \rangle \}$
- $\langle \text{statement} \rangle ::= \langle \text{while loop} \rangle \mid \dots \text{other statements} \dots$
- $\langle \text{statements} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{statements} \rangle \langle \text{statement} \rangle$
- $\langle \text{empty} \rangle ::=$

What BNF cannot do

- BNF can only describe *local* syntax; it cannot refer to another part of the program
 - Example: BNF cannot say that variables must be declared before they are used
- BNF cannot do numeric comparisons
 - Example: BNF cannot say that `int` values must be between `-2147483648` and `2147483647`

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

- Process of parsing data using a grammar
 - Apply rewrite rules to non-terminals on the RHS of an existing rule
 - To match, the derivation must terminate and be composed of terminals only
- Example

Digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Integer \rightarrow **Digit** | **Digit Integer**

- Is 352 an Integer?

Integer \rightarrow Digit Integer \rightarrow 3 Integer \rightarrow

3 Digit Integer \rightarrow 3 5 Integer \rightarrow

3 5 Digit \rightarrow 3 5 2

Intermediate formats are called **sentential forms**

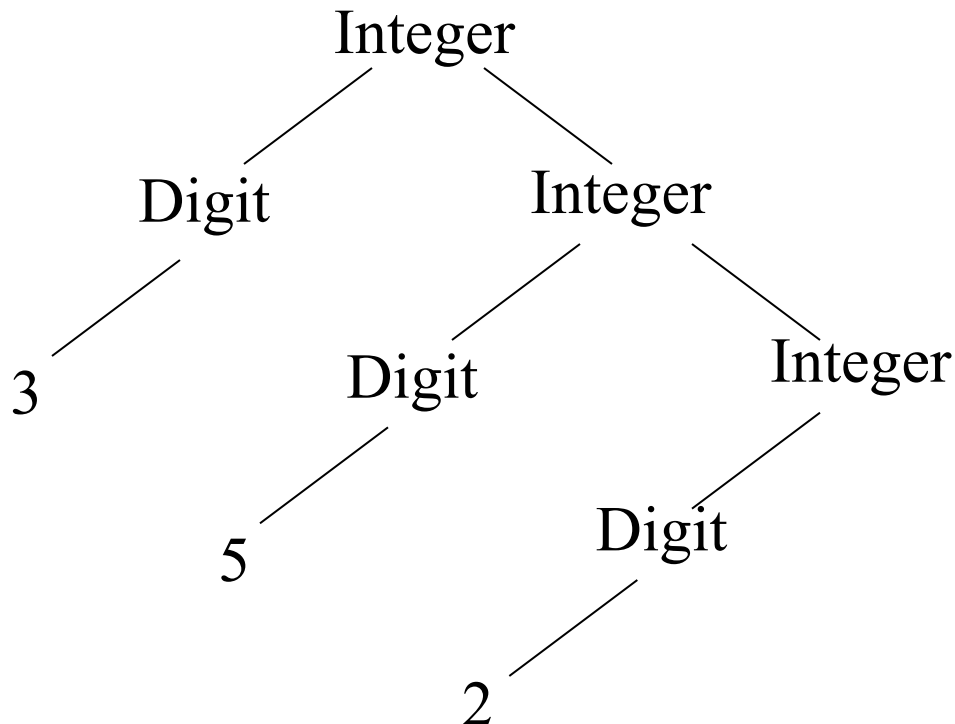
This was called a Leftmost Derivation since we replaced the leftmost nonterminal symbol each time (could also do Rightmost)

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

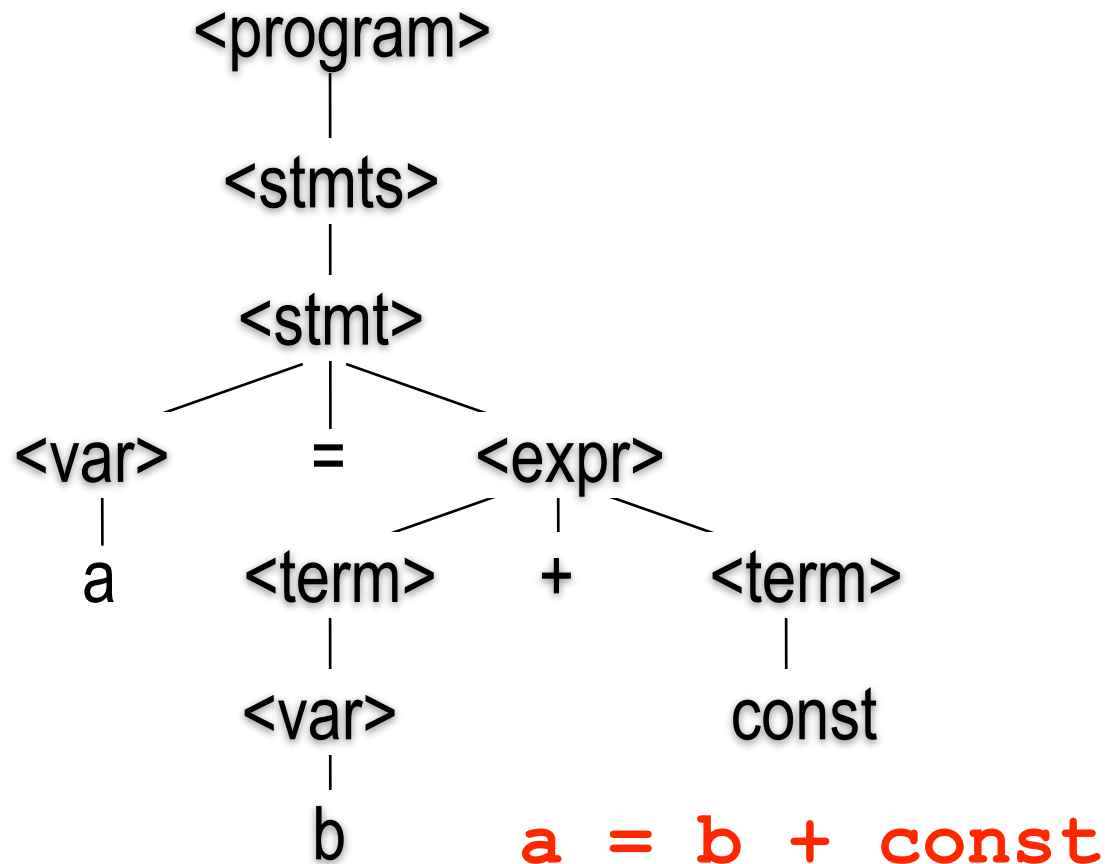
Derivation and Parse Trees

- The derivation can be visualized as a parse tree

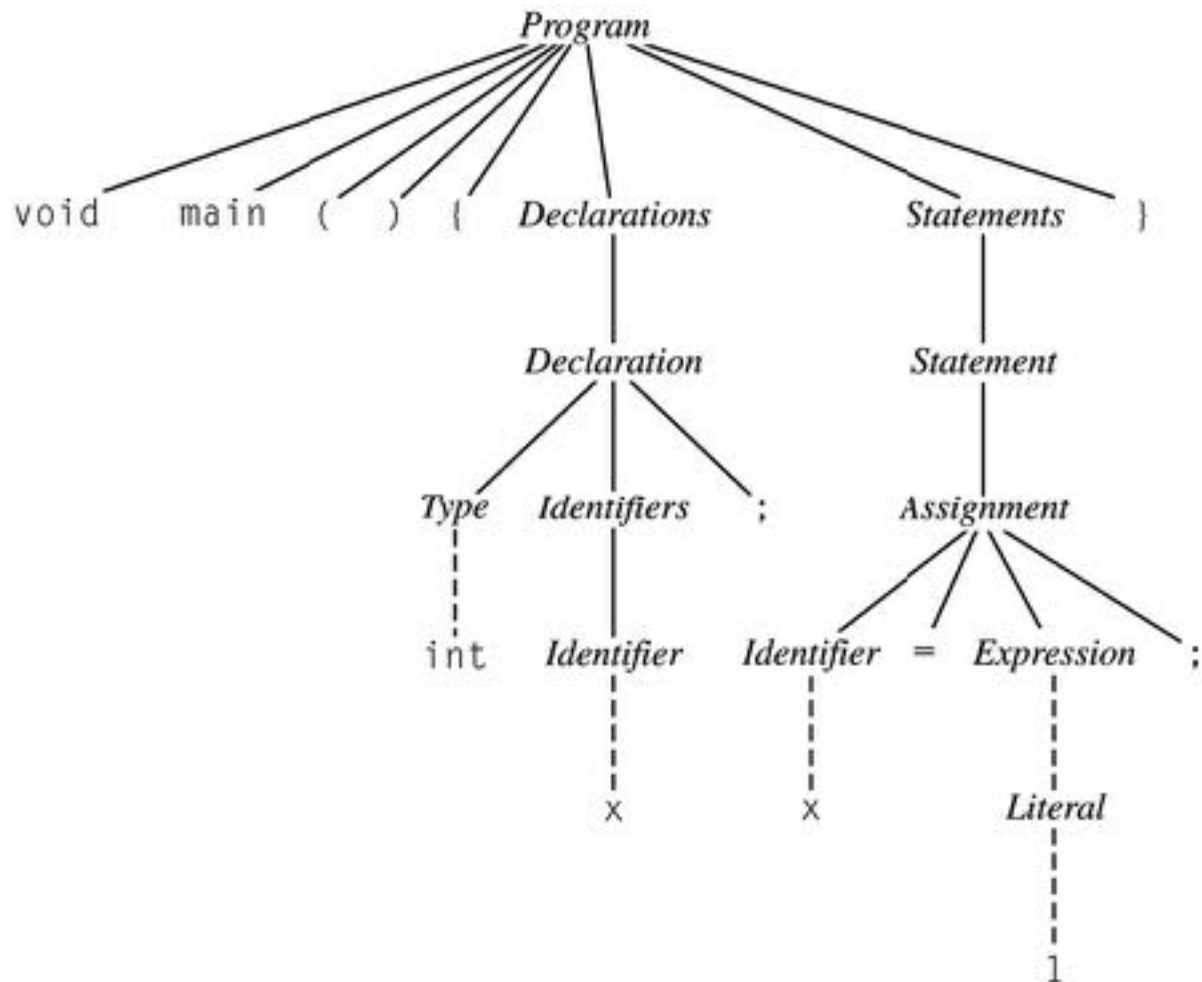


Parse Tree

- ◆ A hierarchical representation of a derivation



Parse Tree Sketch for Programs



Grammar and Parse Tree

- ◆ The grammar can be viewed as a set of rules that say how to build a parse tree
- ◆ You put $\langle S \rangle$ 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

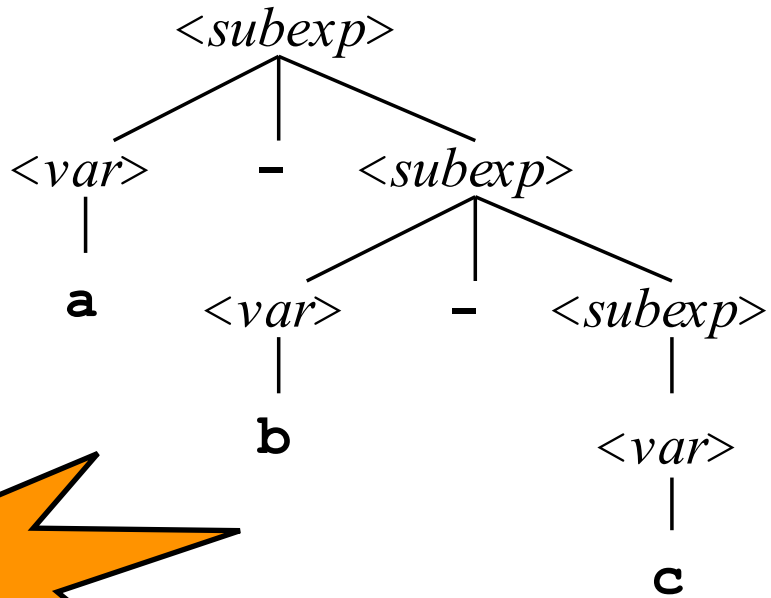
G1: $\langle \text{subexp} \rangle \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \langle \text{subexp} \rangle - \langle \text{subexp} \rangle$

G2: $\langle \text{subexp} \rangle \rightarrow \langle \text{var} \rangle - \langle \text{subexp} \rangle \mid \langle \text{var} \rangle$
 $\langle \text{var} \rangle \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

G3: $\langle \text{subexp} \rangle \rightarrow \langle \text{subexp} \rangle - \langle \text{var} \rangle \mid \langle \text{var} \rangle$
 $\langle \text{var} \rangle \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

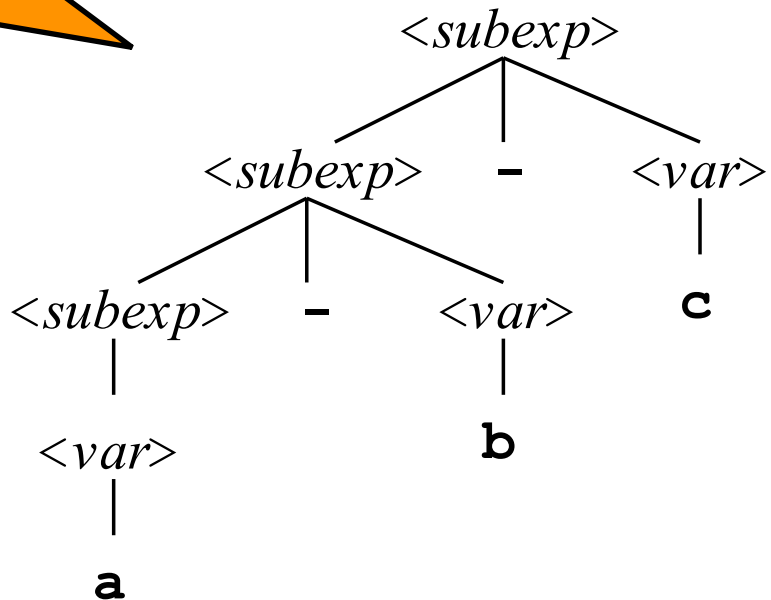
These grammars all define the same language: the language of strings that contain one or more **as**, **bs** or **cs** separated by minus signs, e.g., **a-b-c**. But...

G2 parse tree:



What are
the
differences

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

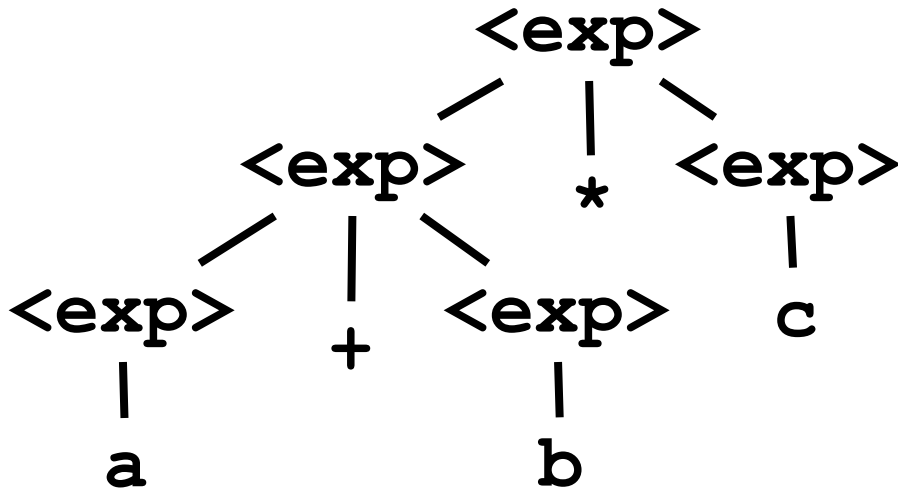
- ◆ Problem with ambiguity:

- Consider the following grammar and the sentence $a+b*c$

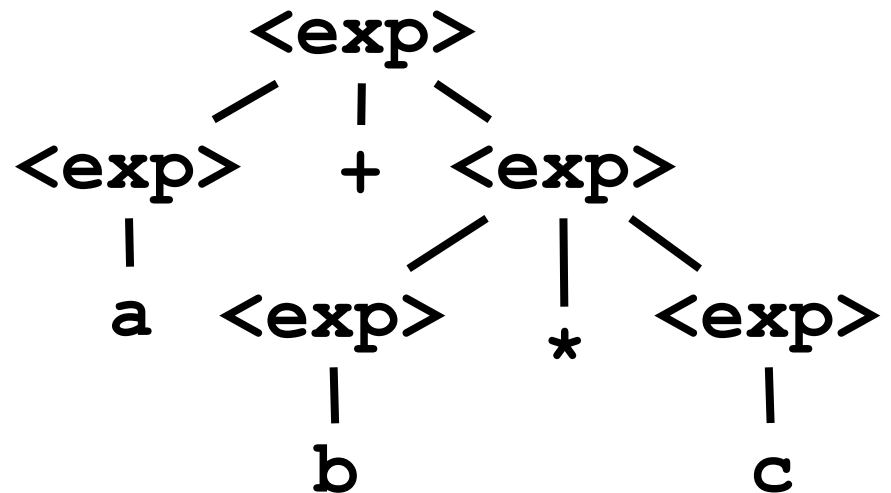
$$\begin{array}{lcl} \langle \text{exp} \rangle & \rightarrow & \langle \text{exp} \rangle + \langle \text{exp} \rangle \\ & | & \langle \text{exp} \rangle * \langle \text{exp} \rangle \\ & | & (\langle \text{exp} \rangle) \\ & | & a \quad | \quad b \quad | \quad c \end{array}$$

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

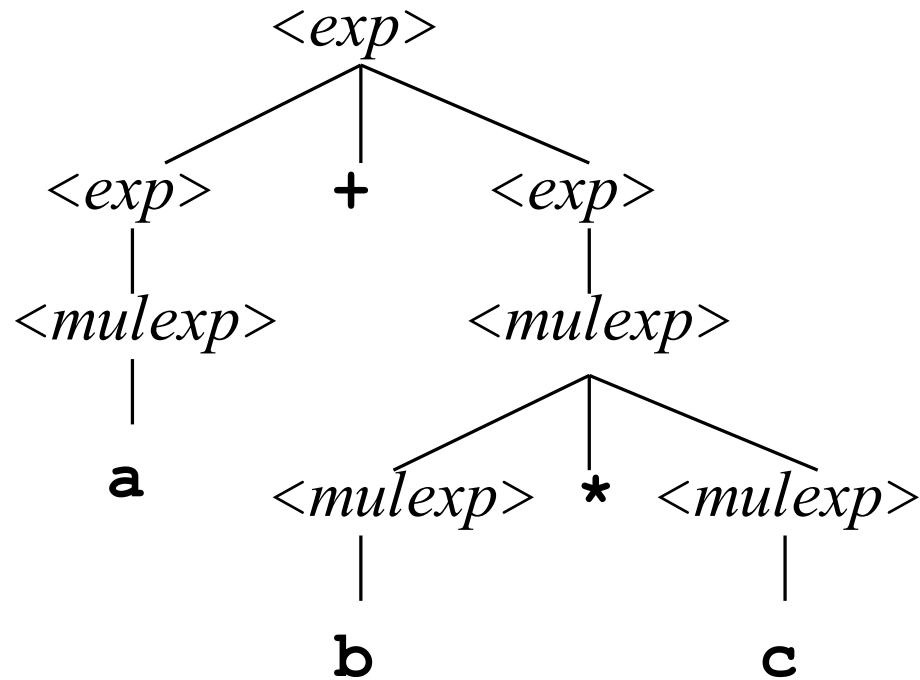
$$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \\ \mid (\langle \text{exp} \rangle) \mid a \mid b \mid c$$

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

$$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{mulexp} \rangle \\ \langle \text{mulexp} \rangle \rightarrow \langle \text{mulexp} \rangle * \langle \text{mulexp} \rangle \\ \mid (\langle \text{exp} \rangle) \mid a \mid b \mid c$$

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.

Ambiguous IF Statement

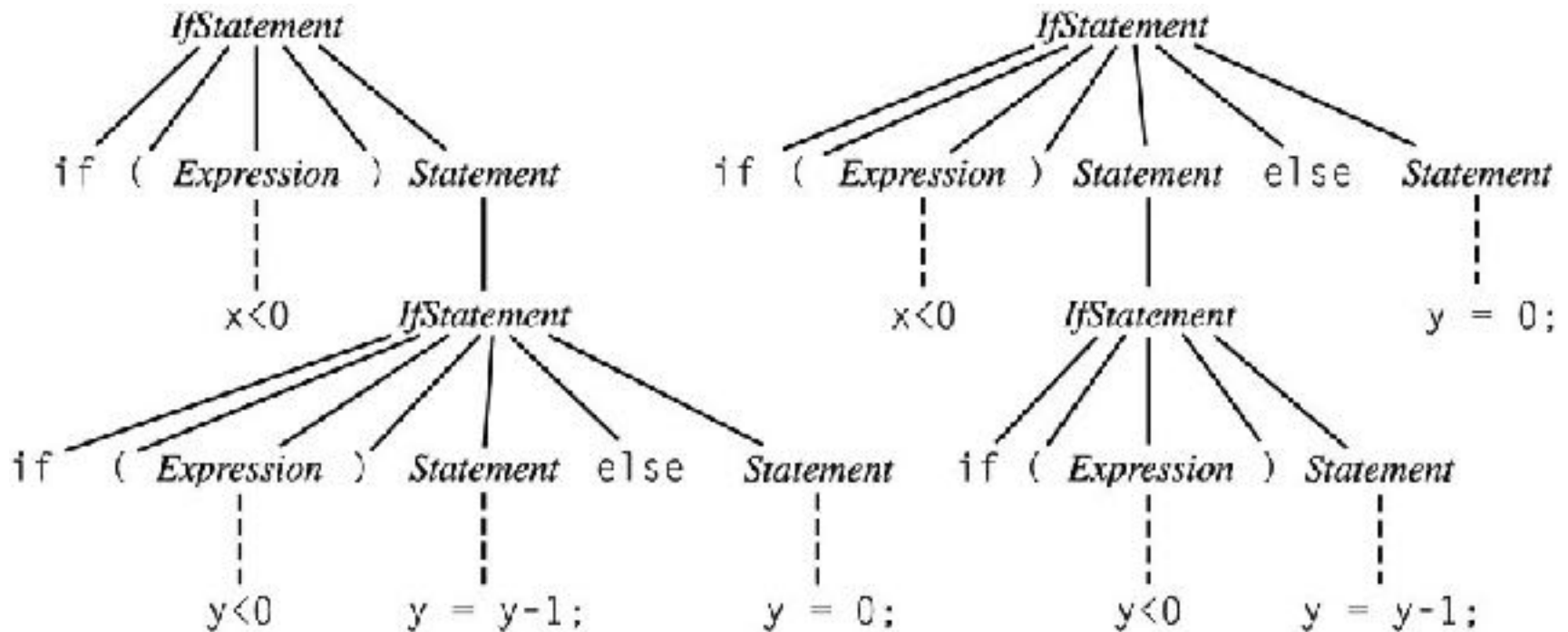
Dangling ELSE:

```
if (x<0)
if (y<0) { y=y-1 }
else { y=0 };
```

Does the else go with the first or second if?

IfStatement \rightarrow if (*Expression*) *Statement* |
if (*Expression*) *Statement* else *Statement*
Statement \rightarrow *Assignment* | *IfStatement*

Dangling Else Ambiguity



How to fix ambiguity?

- Use explicit grammar without ambiguity
 - E.g., add an “ENDIF” for every “IF”
 - Java makes a separate category for if-else vs. if:

IfThenStatement → **If (Expr) Statement**

IfThenElseStatement → **If (Expr) StatementNoShortIf else Statement**

StatementNoShortIf contains everything except **IfThenStatement**, so the else always goes with the **IfThenElse** statement not the **IfThenStatement**

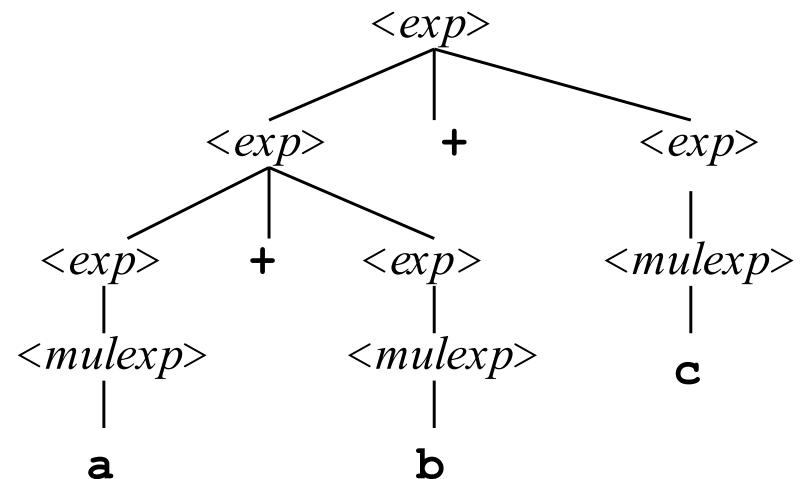
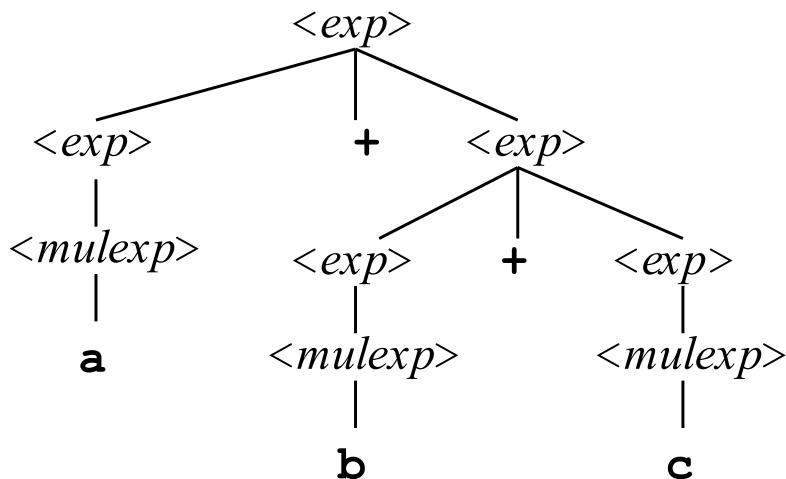
- Use precedence on symbols

Semantics of Associativity

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

$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{mulexp} \rangle$

$\langle \text{mulexp} \rangle \rightarrow \langle \text{mulexp} \rangle * \langle \text{mulexp} \rangle$
 $\mid (\langle \text{exp} \rangle) \mid a \mid b \mid c$



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

$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{mulexp} \rangle$

$\langle \text{mulexp} \rangle \rightarrow \langle \text{mulexp} \rangle * \langle \text{mulexp} \rangle$
 $\mid (\langle \text{exp} \rangle) \mid a \mid b \mid c$

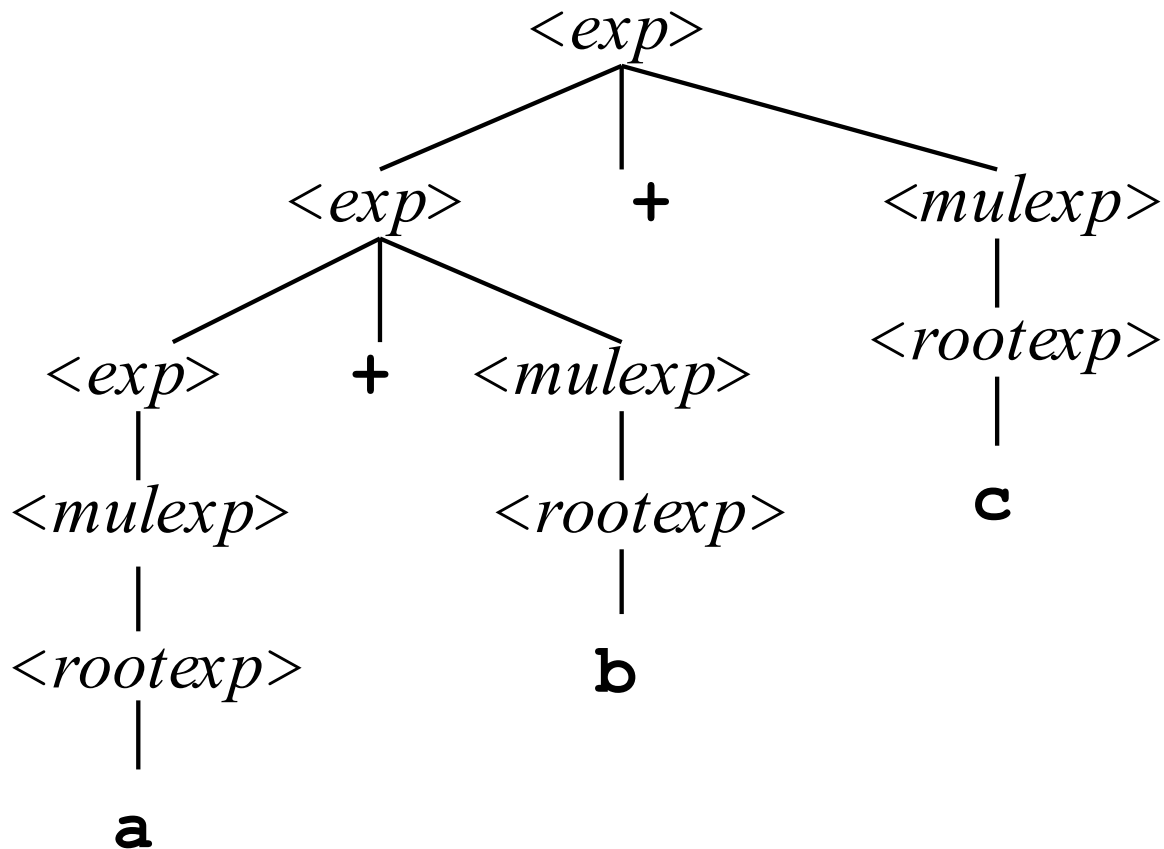
- ◆ To fix the associativity problem, we modify the grammar to make trees of +s grow down to the left (and likewise for *s)

$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle + \langle \text{mulexp} \rangle \mid \langle \text{mulexp} \rangle$

$\langle \text{mulexp} \rangle \rightarrow \langle \text{mulexp} \rangle * \langle \text{rootexp} \rangle \mid$
 $\langle \text{rootexp} \rangle$

$\langle \text{rootexp} \rangle \rightarrow (\langle \text{exp} \rangle) \mid a \mid b \mid c$

Correct Associativity



Dangling Else in Grammars

$\langle stmt \rangle \rightarrow \langle if-stmt \rangle \mid s1 \mid s2$

$\langle if-stmt \rangle \rightarrow \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle$

$\mid \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$

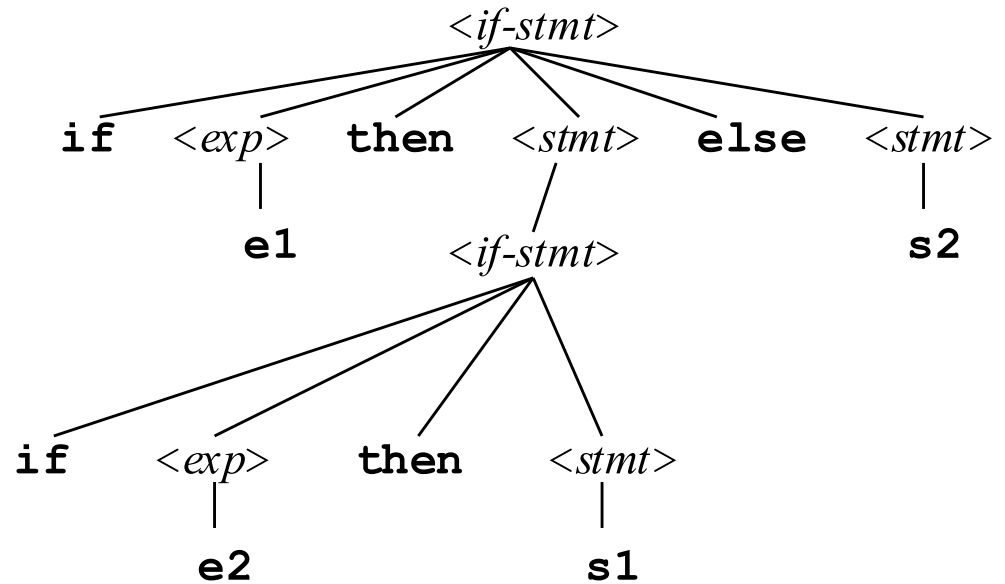
$\langle expr \rangle \rightarrow e1 \mid e2$

- ◆ 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

$\langle stmt \rangle \rightarrow \langle if-stmt \rangle \mid s1 \mid s2$

$\langle if-stmt \rangle \rightarrow \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle$

$\mid \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$

$\langle expr \rangle \rightarrow e1 \mid e2$

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

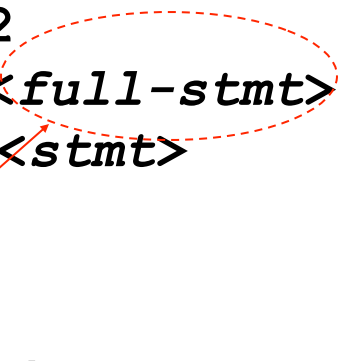
First, we make a new non-terminal $\langle full-stmt \rangle$ that generates everything $\langle stmt \rangle$ generates, except that it can not generate **if** statements with no **else**:

$\langle full-stmt \rangle \rightarrow \langle full-if \rangle \mid s1 \mid s2$

$\langle full-if \rangle \rightarrow \text{if } \langle expr \rangle \text{ then } \langle full-stmt \rangle \text{ else } \langle full-stmt \rangle$

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

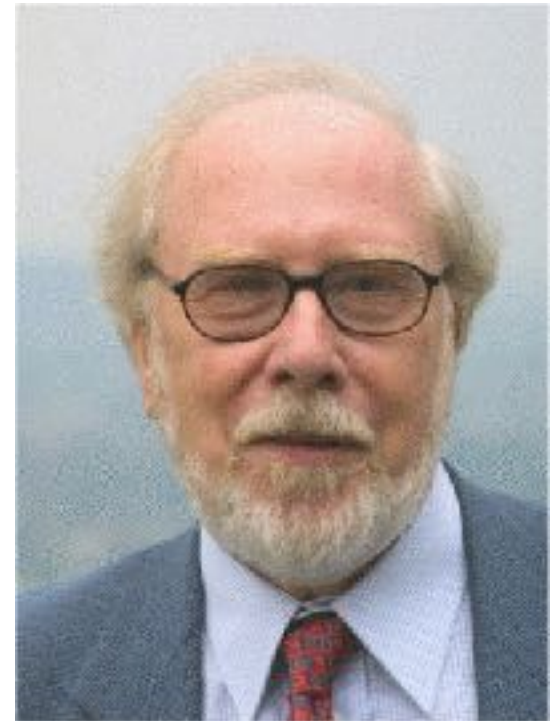
- ◆ Some languages define if-then-else in a way that forces the programmer to be more clear
- ◆ ALGOL does not allow the `then` part to be another `if` statement, though it can be a block containing an `if` statement
- ◆ Ada requires each `if` statement to be terminated with an `end if`

Origin of EBNF

- Stands for “Extended Backus-Naur Form”
- After BNF appeared with Algol 60, lots of people added their own extensions
- Niklaus Wirth wanted to see one form, so he published “What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions” in Communications of the ACM in 1977
- He suggested the use of “[..]” for optional symbols (0 or 1 occurrences), “{ .. }” for 0 or more occurrences.
- Did not mention “EBNF” or Kleene cross

Who was Niklaus Wirth?

- Born in Switzerland, 1934
- Designed Pascal (1970), Modula, Modula-2 (1980), and Oberon (1988)
- Won the Turing Award in 1984
- Won the IEEE Computer Pioneer Award in 1987



What is EBNF?

- EBNF is a few simple extensions to BNF which make expressing grammars more convenient; adds “syntactic sugar”
- Thus it is more concise than BNF
- EBNF is no more “powerful” than BNF; that is, anything that can be expressed in EBNF can also be expressed in BNF
- EBNF is widely used as the de facto standard to define programming languages

What are the Extensions?

- They vary, but often are derived from regular expression syntax
- “*” (The Kleene Star): means 0 or more occurrences
- “+” (The Kleene Cross): means 1 or more occurrences
- “?”: means 0 or 1 occurrences (sometimes “[...]” used instead)
- Use of parentheses for grouping

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

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &\quad | \langle \text{expr} \rangle - \langle \text{term} \rangle \\ &\quad | \langle \text{term} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \\ &\quad | \langle \text{term} \rangle / \langle \text{factor} \rangle \\ &\quad | \langle \text{factor} \rangle\end{aligned}$$

◆ EBNF

$$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$$
$$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$$

BNF vs EBNF

- Grammar for decimal numbers in plain BNF:

`<expr> ::= '-' <num> | <num>`

`<num> ::= <digits>
 | <digits> '.' <digits>`

`<digits> ::= <digit>
 | <digit> <digits>`

`<digit> ::= '0' | '1' | '2' | '3' |
 '4' | '5' | '6' | '7' | '8' | '9'`

BNF vs EBNF

- Same grammar in EBNF:

`<expr> := '-'? <digit>+ ('.' <digit>+)?`

`<digit> := '0' | '1' | '2' | '3' | '4' |
'5' | '6' | '7' | '8' | '9'`

- So much more concise!
- An optional '-', one or more digits, an optional decimal point followed by one or more digits

Simple Conversions

- If you have a rule such as:

`<expr> ::= <digits>`

`<digits> ::= <digit>`

`| <digit> <digits>`

- You can replace it with:

`<expr> ::= <digit>+`

More Simple Conversions

- If you have a rule such as:

`<expr> ::= <digits> | empty`

- You can replace it with:

`<expr> ::= <digit>*`

More Simple Conversions

- If you have a rule such as:

```
<id> ::= <letter>
        | <id><letter>
        | <id><digit>
```

- You can replace it with:

```
<id> ::= <letter> (<letter> | <digit>)*
```

EBNF for Lisp

```
s_expression ::= atomic_symbol  
               | "(" s_expression "." s_expression ")"  
               | list  
list ::= "(" s_expression* ")"  
atomic_symbol ::= letter atom_part  
atom_part ::= empty  
              | letter atom_part  
              | number atom_part  
letter ::= "a" | "b" | " ..." | "z"  
number ::= "1" | "2" | " ..." | "9"
```

Summary-BNF

- BNF uses following notations:
 - (i) Non-terminals enclosed in $<$ and $>$.
 - (ii) Rules written as
$$X ::= Y$$
 - (1) X is LHS of rule and can only be a NT.
 - (2) Y is RHS of rule: Y can be
 - (a) a terminal, nonterminal, or concatenation of terminal and nonterminals, or
 - (b) a set of strings separated by alternation symbol $|$.

Example:

$<S> ::= a <S> | a$

- Notation ϵ : Used to represent an empty string (a string of length 0).

Extended BNF (EBNF)

- EBNF: adding more meta-notation \Rightarrow shorter productions
- NTS begin with uppercase letters (discard $\triangleleft \triangleright$)
- Repetitions (zero or more) are enclosed in $\{ \}$
- Zero or one (options) are enclosed in $[]$:
 $\text{Ifstmt} ::= \text{if Cond then Stmt} \mid$
 $\text{if Cond then Stmt else Stmt}$
 $\Rightarrow \text{Ifstmt} ::= \text{if Cond then Stmt} [\text{else Stmt}]$
- Use $()$ to group items together:
 $\text{Exp} ::= \text{Item} \{ + \text{Item} \} \mid \text{Item} \{ - \text{Item} \}$
 $\Rightarrow \text{Exp} ::= \text{Item} \{ (+|-) \text{Item} \}$
- Terminals that are grammar symbols ($'[$ ' for instance) are enclosed in quotes (") .

Summary

Conversion from EBNF to BNF and Vice Versa

· BNF to EBNF:

(i) Look for recursion in grammar:

$$A ::= a A \mid B$$
$$\Rightarrow A ::= a \{ a \} B$$

(ii) Look for common string that can be factored out with grouping and options.

$$A ::= a B \mid a$$
$$\Rightarrow A ::= a [B]$$

· EBNF to BNF:

(i) Options: $[]$

$$A ::= a [B] C$$
$$\Rightarrow A' ::= a N C \quad N ::= B \mid \varepsilon$$

(ii) Repetition: $\{\}$

$$A ::= a \{ B_1 B_2 \dots B_n \} C$$
$$\Rightarrow A' ::= a N C \quad N ::= B_1 B_2 \dots B_n N \mid \varepsilon$$