CHAPTER TWO

Syntax and Semantic

Chapter 2 Topics

- Introduction
- Organization of Language Description
- Describing Syntax
- Formal Methods of Describing Syntax
- The Way of Writing Grammars
- Formal Semantic
- Semantic

Introduction

- Who must use language definitions?
  - Other language designers
  - Implementers
  - Programmers (the users of the language)
- Syntax - the form or structure of the expressions, statements, and program units
- Semantics - the meaning of the expressions, statements, and program units

Introduction

- Language description
  - → syntax and semantic
- Syntax
  - → how to write program
- Semantic:
  - → what does program mean

Introduction

- Dates represented by
  D (digit) and Symbol (/)

  DD/DD/DDDD -> syntax
  01/02/2001 -> US Jan 2, 2001
  Others Feb 1, 2001

- Same syntax, different semantic

Organization of Language Description

- Tutorials
- Reference Manuals
- Formal Definition

Tutorials

- What the main constructs of the language are
- How they are meant to be used
- Useful examples for imitating and adapting
- Introduce syntax and semantics gradually

Reference Manuals

- Describing the syntax and semantics
- Organized around the syntax
- Formal syntax and informal semantic
- Informal semantic : English explanations and examples to the syntactic rules
- Began with the Algol60 : free of ambiguities

Formal Definition

- Precise description of the syntax and semantics
- Aimed at specialists
- Attaches semantic rules to the syntax
- Conflicting interpretations from English explanation
- Precise formal notation for clarifying subtle point

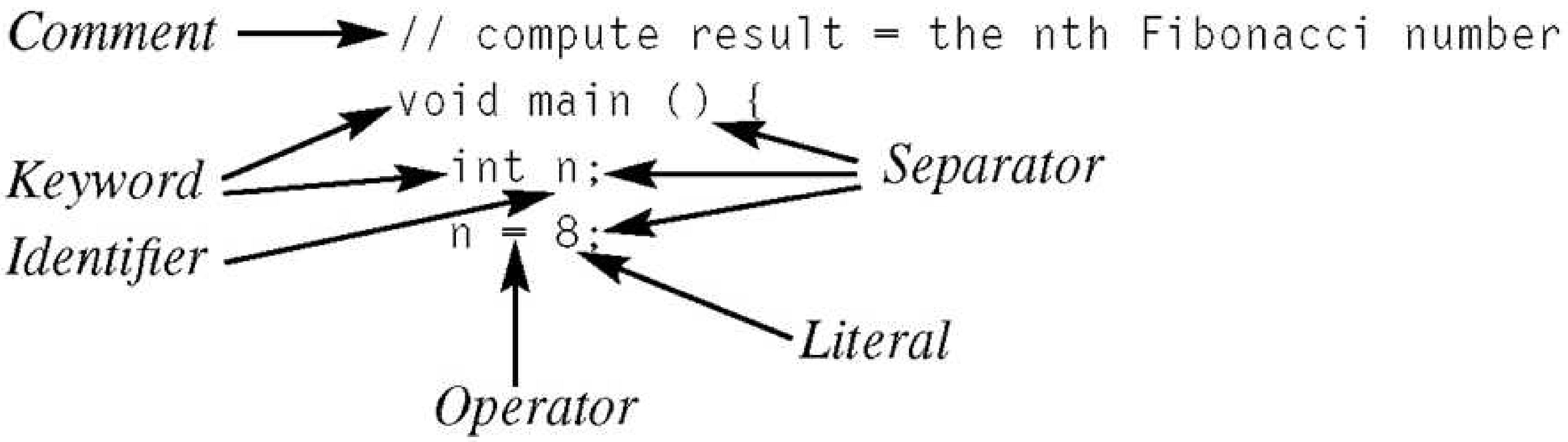
Syntactic Elements

- Character set
- Identifiers
- Operator symbols
- Keywords / Reserved words
- Comments
- Separator & Brackets
- Expression
- Statements

Describing Syntax

- 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)

A Program Fragment Viewed As a Stream of Tokens



```
// compute result = the nth Fibonacci number
void main () {
int n;
n = 8;
```

Describing Syntax

- Formal approaches to describing syntax:
  - Recognizers - used in compilers
  - Generators – generate the sentences of a language (focus of this lecture)

Formal Methods of Describing Syntax

- 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

CFG for Thai

<ประโยค> -> <ประธาน><กริยา><กรรม>

<ประธาน> -> ฉัน | เธอ | เรา

<กริยา> -> กิน | ตี

<กรรม> -> ข้าว| สุนัข

<ประโยค> -> <ประธาน><กริยา><กรรม>

ฉัน กิน ข้าว

เธอ ตี ข้าว

Formal Methods of Describing Syntax

- Backus-Naur Form (1959)
  - Invented by Backus and Naur to describe Algol 58
  - BNF is equivalent to context-free grammars
  - A metalanguage is a language used to describe another language.

Backus-Naur Form (1959)

BNF elements

- T - terminal symbols
- N - nonterminal symbols
- S - start symbol
- P - set of rules or production

BNF grammar

Def: A grammar production has the form

A -> ω   where A is a nonterminal symbol

ω is a string of nonterminal and terminal symbols

- This is a rule; it describes the structure of a while statement

<while_stmt> → `while` (<logic_expr>) <stmt>

Formal Methods of Describing Syntax

- 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`

BNF

- Nonterminal
  - Identifier
  - Integer
  - Expression
  - Statement
  - Program
- Terminal
  - The basic alphabet from which programs are constructed

binaryDigit -> 0
binaryDigit -> 1

binaryDigit -> 0 | 1

Integer -> *Digit* | *Integer Digit*
Digit -> 0|1|2|3|4|5|6|7|7|8|9

Integer -> *Digit*
Integer -> *Integer Digit*
Integer -> *Integer Integer Digit*
Integer -> *Digit Digit*

Formal Methods of Describing Syntax

- 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)

Formal Methods of Describing Syntax

- 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`

Derivation

- **Grammar**

  Integer -> *Digit* | *Integer Digit*

  Digit -> 0|1|2|3|4|5|6|7|7|8|9

- Is 352 an *Integer*?

  Integer => *Integer Digit*
  => *Integer Digit Digit*
  => *Digit Digit Digit*
  => *3 Digit Digit*
  => *35 Digit*
  => *352*

Derivation

- **Grammar**

```
<assign> -> <id> = <expr>
<id>        -> A | B | C
<expr>    -> <id> + <expr>
               | <id> * <expr>
               | ( <expr> )
               | <id>
```

- **Statement**

A = B * (A + C)

- **Leftmost derivation**

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr> )
         => A = B * ( <id> + <expr>)
         => A = B * ( A + <expr>)
         => A = B * ( A + <id>)
         => A = B * ( A + C )
```

Formal Methods of Describing Syntax

- An example derivation:

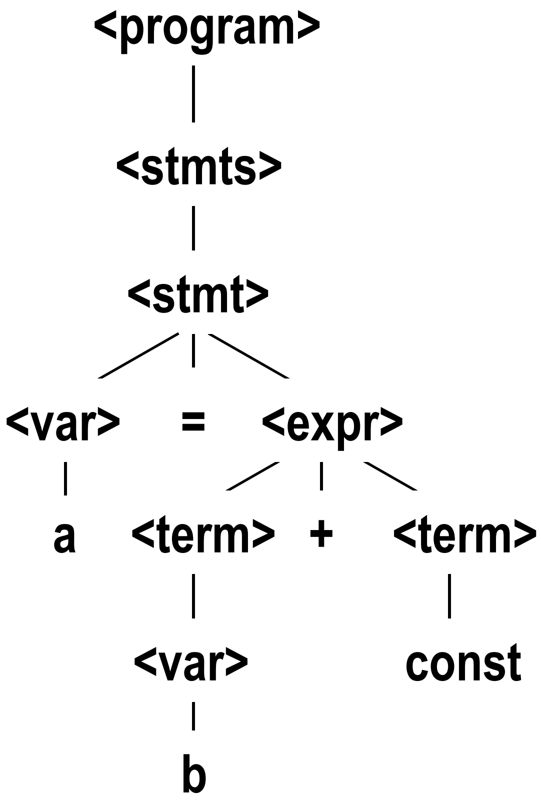
```
<program>  => <stmts>
           => <stmt>
           => <var> = <expr>
           => a = <expr>
           => a = <term>  +  <term>
           => a = <var>  +  <term>
           => a  =  b  +  <term>
           => a  =  b  +  const
```

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

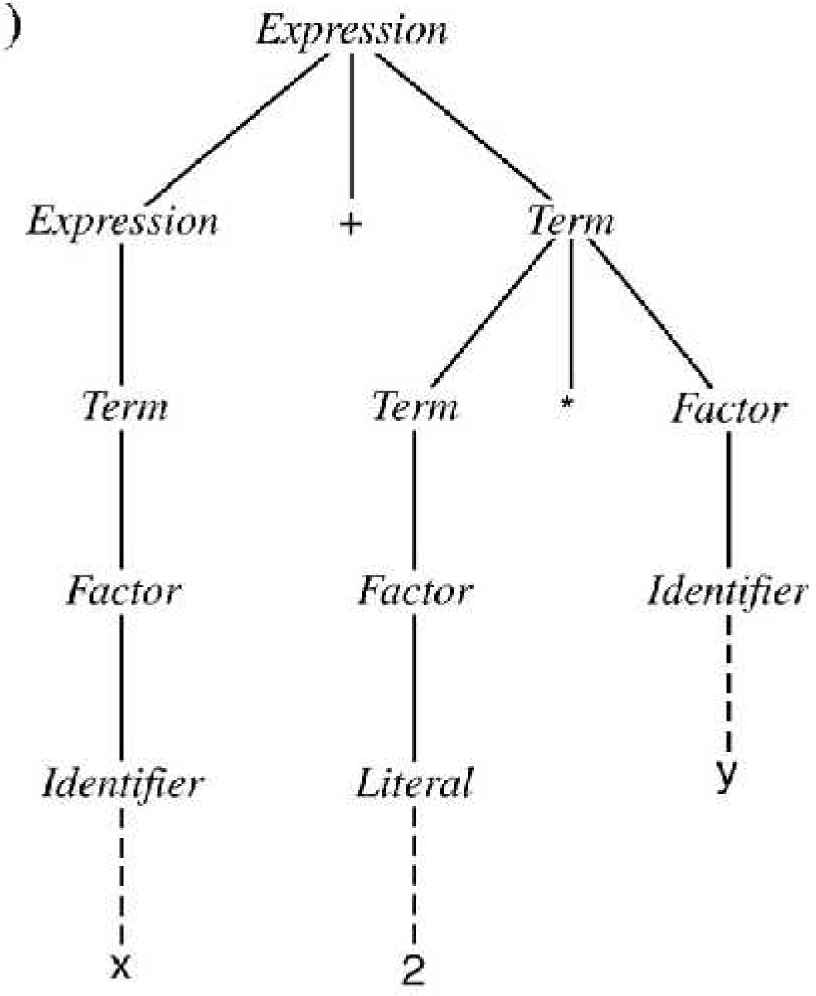
Parse Tree

- A hierarchical representation of a derivation



Parse Tree for the *Expression* x+2*y

Assignment → *Identifier* = *Expression* ;
Expression → *Term* | *Expression* + *Term* | *Expression* − *Term*
Term → *Factor* | *Term* * *Factor* | *Term* / *Factor*
Factor → *Identifier* | *Literal* | (*Expression*)

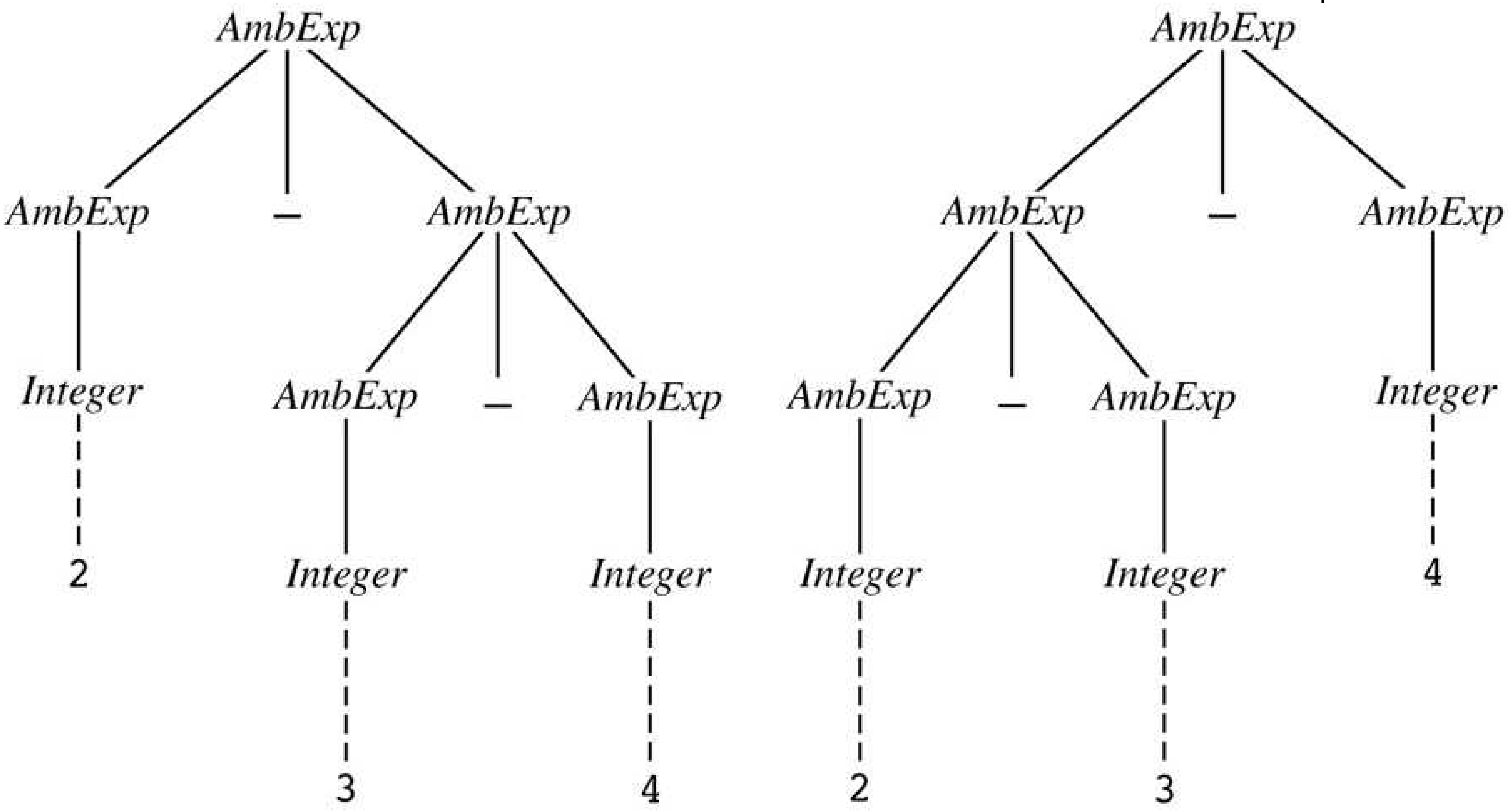


Formal Methods of Describing Syntax

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

An Ambiguous Grammar

<AmbExp> → <Integer> | <AmbExp> - <AmbExp>



Two Different Parse Trees for the *AmbExp* 2 – 3 – 4

Is the Grammar Ambiguous?

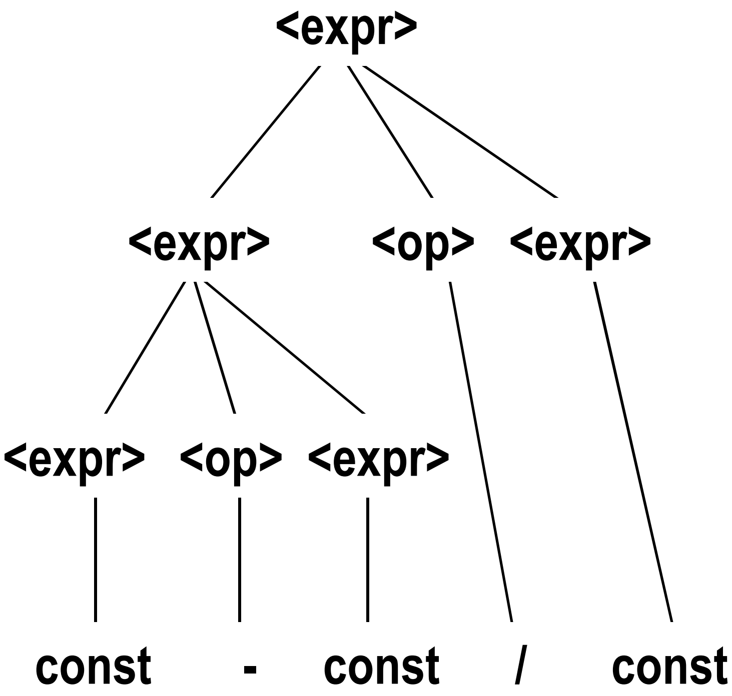
<expr> → <expr> <op> <expr> | const

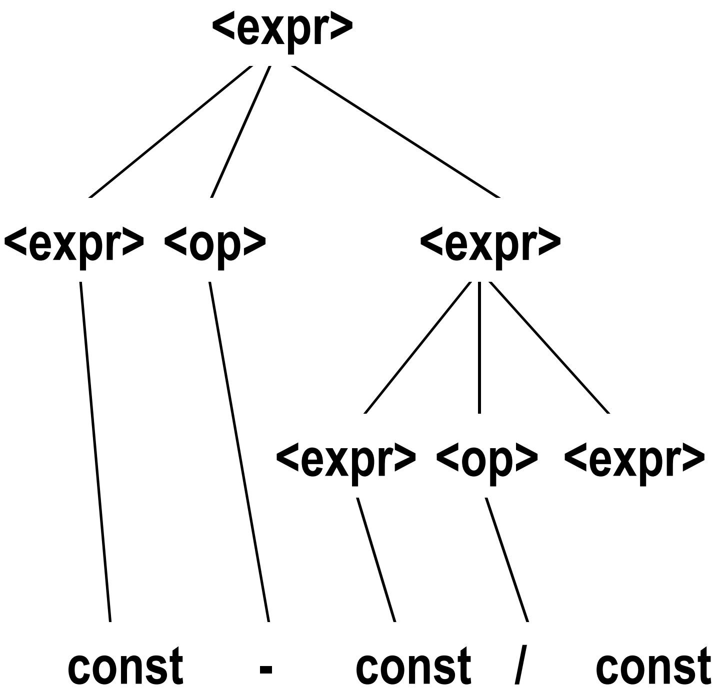
<op> → / | -

Is the Grammar Ambiguous? Yes

<expr> → <expr> <op> <expr> | const

<op> → / | -

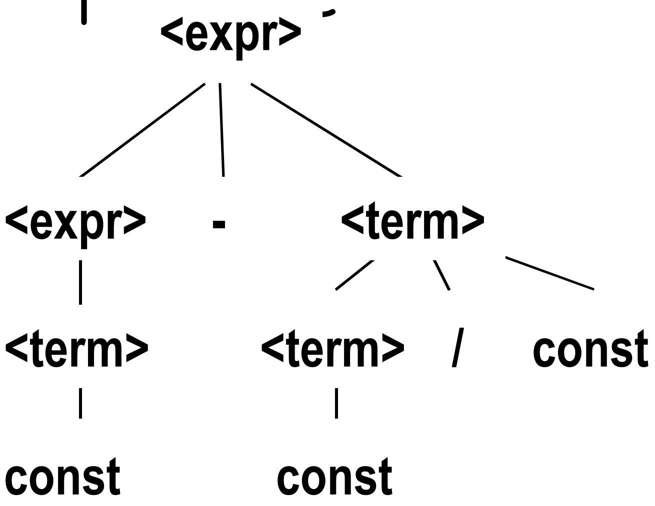




An Unambiguous Expression Grammar

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

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



<expr>

<expr> - <term>

<term> <term> / const

const const

Formal Methods of Describing Syntax

Derivation:

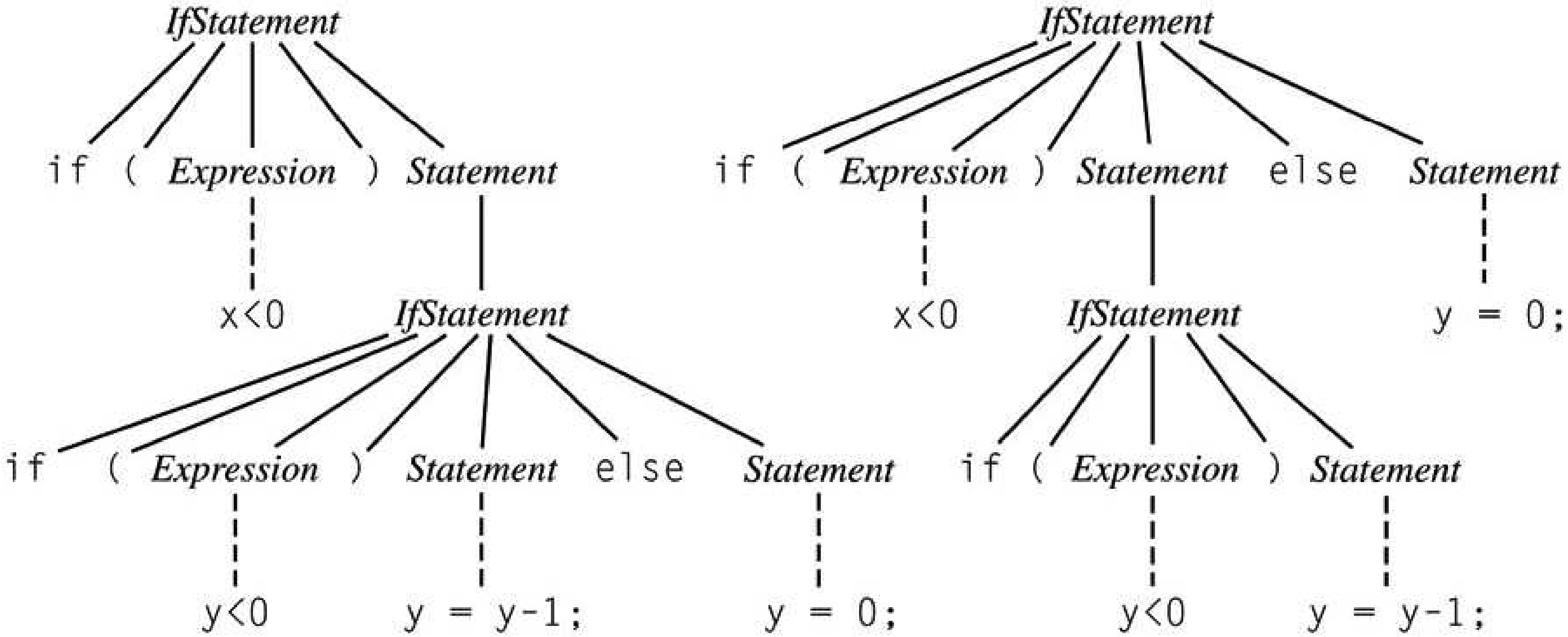
<expr> => <expr> - <term> => <term> - <term>
=> `const` `-` **<term>**
=> `const` `-` **<term>** `/ const`
=> `const - const / const`

An Ambiguous If Statement

The “Dangling Else” Grammatical Ambiguity

IfStatement → if (*Expression*) *Statement* |
if (*Expression*) *Statement* else *Statement*

Statement → *Assignment* | *IfStatement*



Formal Methods of Describing Syntax

- Extended BNF (just abbreviations):
- Notation used in the course textbook
  - Optional parts:

<proc_call> -> `ident` **(<expr_list>)$_{\mathbf{opt}}$**

  - Alternative parts:

<term> -> <term> `[+ | -] const`

  - Put repetitions (0 or more) in braces ({ })

<ident> -> `letter {letter | digit}*`

Formal Methods of Describing Syntax

- Extended BNF (just abbreviations):
- Another frequently used notation
  - Optional parts:

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

  - Alternative parts:

<term> -> <term> `(+ | -) const`

  - Put repetitions (0 or more) in 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>}*

The Way of Writing Grammars

- The productions are rules for building string
- *Parse Trees* : show how a string can be built

Notation to write grammar

- Backus-Naur Form (BNF)
- Extended BNF (EBNF)
- Syntax charts : graphical notation

BNF

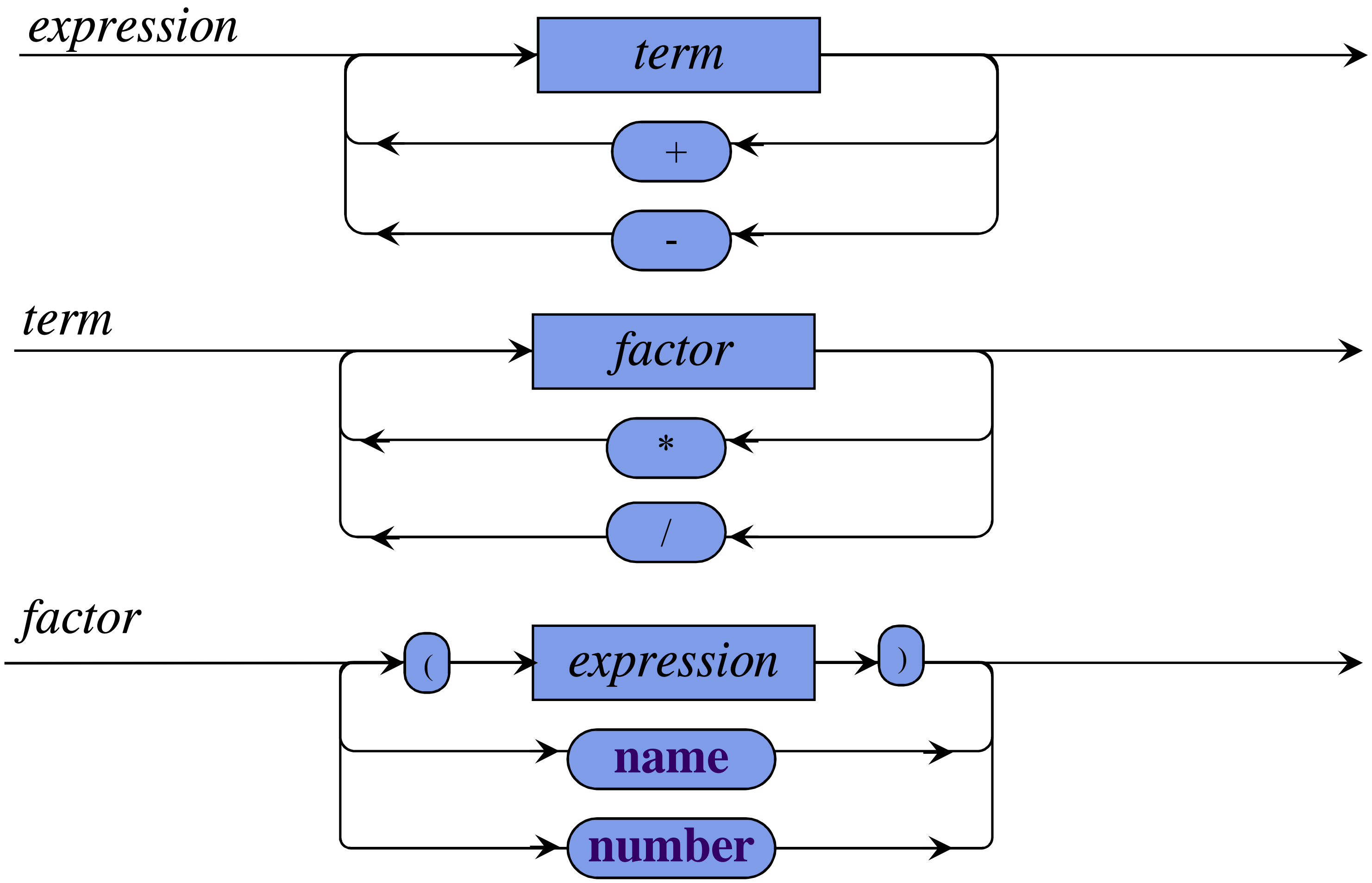
```
<expression > ::=  <expression> + <term >
                   |  <expression> - <term>
                   |  <term>
<term >        ::=  <term> * <factor>
                   |  <term> / <factor>
                   |  <factor>
<factor >      ::=  number
                   |  name
                   |  ( <expression> )
```

Extended BNF

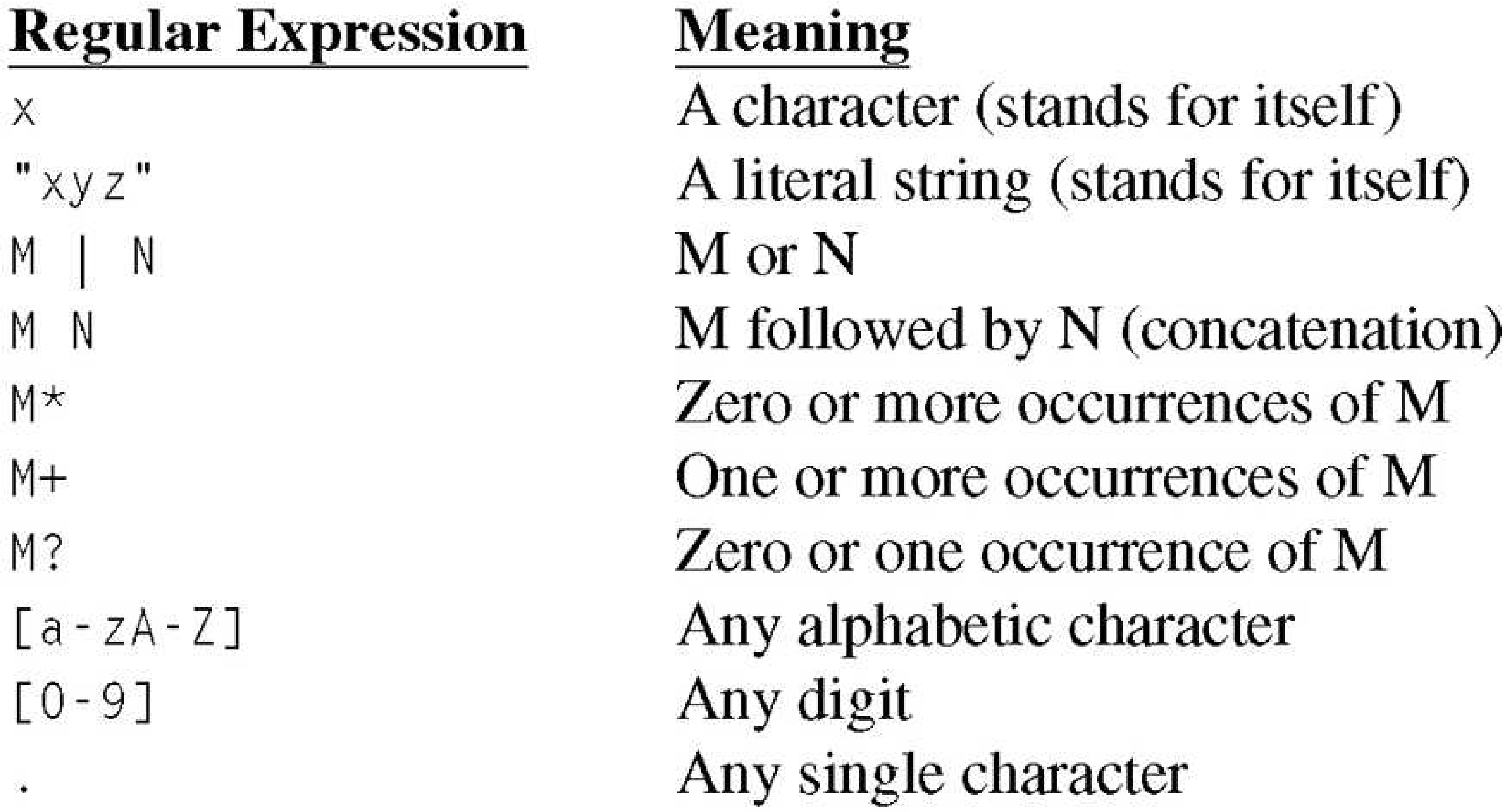
<expression > ::= <term> { (+ | -) <term> }
<term > ::= <factor> { (* | /) <factor> }
<factor > ::= (<expression>)
| name
| number

Syntax Diagram

- Can be used to visualize rules



Conventions for Writing Regular Expressions



Regular Expression	Meaning
x	A character (stands for itself)
"xyz"	A literal string (stands for itself)
M | N	M or N
M N	M followed by N (concatenation)
M*	Zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[a-zA-Z]	Any alphabetic character
[0-9]	Any digit
.	Any single character

Semantics or meaning

- Semantic : any property of a construct
- The semantic of an expression 2+3

Point of view	Semantic
An expression evaluator	its value : 5
A type checker	type integer
An infix-to-postfix translator	string: + 2 3

Formal Semantic

- Static semantic : “***compile-time***” properties
  - type correctness, translation
  - determined from the static text of a program,
  without running the program on actual data.
- Dynamic semantic : “***run-time***” properties
  - value of expression
  - the effect of statement
  - determined by actually doing a computation

Semantic Methods

- Several method for defining semantics
- The approaches are used for different purposes by different communities.
- The values of variables a and b in a+b depend on the environment.
- An environment consists of bindings from variable to values.

Assignment

- Draw a parse tree using BNF grammar on slide page 40
  - 2 + 3
  - (2 + 3)
  - 2 + 3 * 5
  - (2 + 3) * 5
  - 2 + (3 * 5)

Assignment (cont.)

- Draw parse trees using the following grammar

```
S ::= id := expr
      | if expr then S
      | if expr then S
        else S
      |  while expr do S
      |  begin SL end
SL ::=   S ;
      |   S ;  SL
```

- while expr do id := expr
- begin id := expr end
- if expr then if expr then S else S

Assignment (cont.)

- Write the grammar in any language by using BNF or EBNF or syntax diagram
- Write the keywords in that languages
- Deadline : next class

References

- Books
  - Programming Languages: Principles and Paradigms – Allen B. Tucker & Robert E. Noonan
  - Concepts of Programming languages – Robert W. Sebesta

- Java BNF
  - http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html