Chapter 4 - Syntax

Programming Languages:

Principles and Practice, 2nd Ed.

Kenneth C. Louden

Supplementary Material-1 (SacCT)

- **CFG /BNF, EBNF, Syntax Diagram**

- **First and Follow sets**

Supplementary Material-2

- **Recursive Descent**
  - **Dr. Gordon's example (hardcopy)**
  - **PHP implementation examples (transparency slides)**

Introduction

- **Syntax is the *structure* of a language, i.e., the form that each program or source code file must take.**
- **Since the early 1960s, syntax has been given as a set of *grammar rules* in a form developed by Noam Chomsky, John Backus, and Peter Naur. (Context-free grammar, Backus Naur Form [BNF].)**
- **Syntax includes the definition of the words, or *tokens*, of the language, which can be called its *lexical structure*.**
- **Both lexical and syntactic structure have precise mathematical definitions that every computer scientist should know.**

Lexical Structure

- **Tokens are the building blocks of syntax: the "words" of the language.**
- **Tokens are recognized by the first phase of a translator -- the scanner -- which is the only part of the translator that deals directly with the input. The scanner groups the input characters into tokens. (Scanners are sometimes called lexers.)**
- **Tokens can be defined using either grammar rules or *regular expressions*.**
- **Regular expressions are used in many areas of computer science to describe text patterns. One example: grep ("global regular expression print").**
- **Sample regular expression:** `[0-9]+(\.[0-9]+)?`

Standard Token Categories

- **Reserved words, sometimes called *keywords*, such as `if` and `while`**
- **Literals or constants, such as `42` (a numeric literal) or `"hello"` (a string literal)**
- **Special symbols, such as “`;`”, “`<=`”, or “`+`”**
- **Identifiers, such as `x24`, `monthly_balance`, or `putchar`**

White space and comments

- **"Internal" tokens of the scanner that are matched and discarded**
- **Typical white space: newlines, tabs, spaces**
- **Comments:**
  - **`/*` … `*/`, `//` … `\n` (C, C++, Java)**
  - **`--` … `\n` (Ada, Haskell)**
  - **`(*` … `*)` (Pascal, ML)**
  - **`;` … `\n` (Scheme)**
- **Comments generally not nested.**
- **Comments & white space ignored except they function as *delimiters* (or *separators*).**

Reserved words versus predefined identifiers

- **Reserved words cannot be used as the name of anything in a definition (i.e., as an identifier).**
- **Predefined identifiers have special meanings, but can be redefined (although they probably shouldn't).**
- **Examples of predefined identifiers in Java: anything in java.lang package, such as String, Object, System, Integer.**

Java reserved words

`abstract`	`default`	`if`	`private`	`this`
`boolean`	`do`	`implements`	`protected`	`throw`
`break`	`double`	`import`	`public`	`throws`
`byte`	`else`	`instanceof`	`return`	`transient`
`case`	`extends`	`int`	`short`	`try`
`catch`	`final`	`interface`	`static`	`void`
`char`	`finally`	`long`	`strictfp`	`volatile`
`class`	`float`	`native`	`super`	`while`
`const`	`for`	`new`	`switch`	
`continue`	`goto`	`package`	`synchronized`	

Java reserved words (cont.)

- **The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.**
- **While `true` and `false` might appear to be keywords, they are technically boolean literals (§3.10.3). Similarly, while `null` might appear to be a keyword, it is technically the null literal (§3.10.7).**

C tokens

"There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments as described below (collectively, "white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token." [Kernighan and Ritchie, *The C Programming Language*, 2nd Ed., pp. 191-192.]

- **Note principle of longest substring (true for virtually every language except FORTRAN): `ifelse` is an identifier, not two keywords.**

Scanners

- **Recognizers of regular expressions**
- **Implemented as finite automata, a.k.a. finite state machines**
- **Typically contain a loop that cycles through characters, building tokens and associated values by repeated operations**
- **This process is repeated for each token**
- **A single token recognition is encapsulated in a getToken() (or similarly named) procedure**

Simple scanner example

- **Tokens from integer arithmetic:**
  + - * / Number ([0-9]+), left & right parens, and white space, all on a single line of input
- **EOL token also needed (to end input)**
- **Error token also needed (for illegal characters)**
- **C code is in Figure 4.1, pages 82-83**
  (only + and * are implemented in that code)
- **Input taken from standard input (keyboard or redirected file)**
- **End of file check not implemented**

Context-free grammars

Figure 4.2, page 83:

(1) *sentence* → *noun-phrase* *verb-phrase* **.**

(2) *noun-phrase* → *article* *noun*

(3) *article* → **a** | **the**

(4) *noun* → **girl** | **dog**

(5) *verb-phrase* → *verb* *noun-phrase*

(6) *verb* → **sees** | **pets**

Terminology

- **Left-hand sides (before the $\rightarrow$) are called nonterminals or structure names.**
- **Right-hand sides (after the $\rightarrow$) are strings of tokens and nonterminals, sometimes called symbols. (Metasymbols with special meanings can also sometimes appear.)**
- **Tokens are sometimes called terminals.**
- **Grammar rules themselves are sometimes called productions, since they "produce" the language.**
- **Metasymbols are the arrow $\rightarrow$ (“consists of”) and the vertical bar | (choice).**
- **One nonterminal is singled out as the start symbol: it stands for a complete unit in the language (sentence, program).**

CFGs generate "languages":

- **The language of a CFG is the set of strings of terminals that can be generated from the start symbol by a derivation:**

  sentence ⇒ *noun-phrase verb-phrase* **.** (rule 1)

  ⇒ *article noun verb-phrase* **.** (rule 2)

  ⇒ **the** *noun verb-phrase* **.** (rule 3)

  ⇒ **the girl** *verb-phrase* **.** (rule 4)

  ⇒ **the girl** *verb noun-phrase* **.** (rule 5)

  ⇒ **the girl sees** *noun-phrase* **.** (rule 6)

  ⇒ **the girl sees** *article noun* **.** (rule 2)

  ⇒ **the girl sees a** *noun* **.** (rule 3)

  ⇒ **the girl sees a dog .** (rule 4)

CFGs can be recursive:

expr → *expr* **+** *expr* | *expr* ***** *expr*
| **(** *expr* **)** | *number*

number → *number digit* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

or

expr → *expr* **+** *expr* | *expr* ***** *expr*
| **(** *expr* **)** | **NUMBER**

NUMBER = [0-9]+

Notes:

- **Recursion can be used to get simple repetition (Example 1):**

  number $\Rightarrow$ *number digit* $\Rightarrow$ *number digit digit*

  $\Rightarrow$ *digit digit digit* $\Rightarrow$ **2** *digit digit*

  $\Rightarrow$ **23** *digit* $\Rightarrow$ **234**

- **Recursion is more powerful than just simple repetition (Example 2):**

  expr $\Rightarrow$ *expr* ***** *expr* $\Rightarrow$ **(** *expr* **)** ***** *expr*

  $\Rightarrow$ **(** *expr* **+** *expr* **)** ***** *expr* $\Rightarrow$ ...

  $\Rightarrow$ **(2 + 3) * 4**

Parse Trees

- **Derivations express the structure of syntax, but not very well: there can be lots of different derivations for the same structure, e.g. Example 1 could have been:**

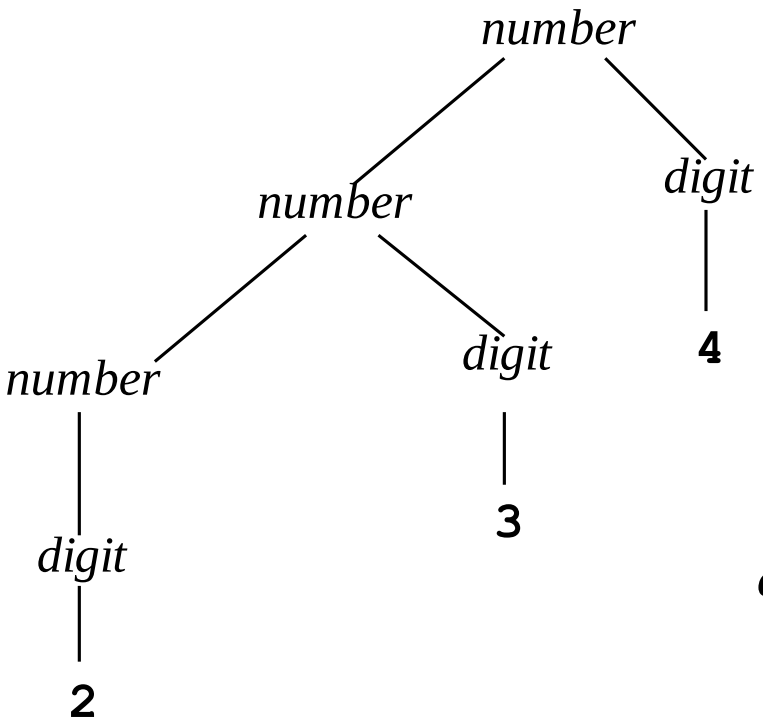
  number $\Rightarrow$ *number digit* $\Rightarrow$ *number* `4`

  $\Rightarrow$ *number digit* `4` $\Rightarrow$ *number* `34`

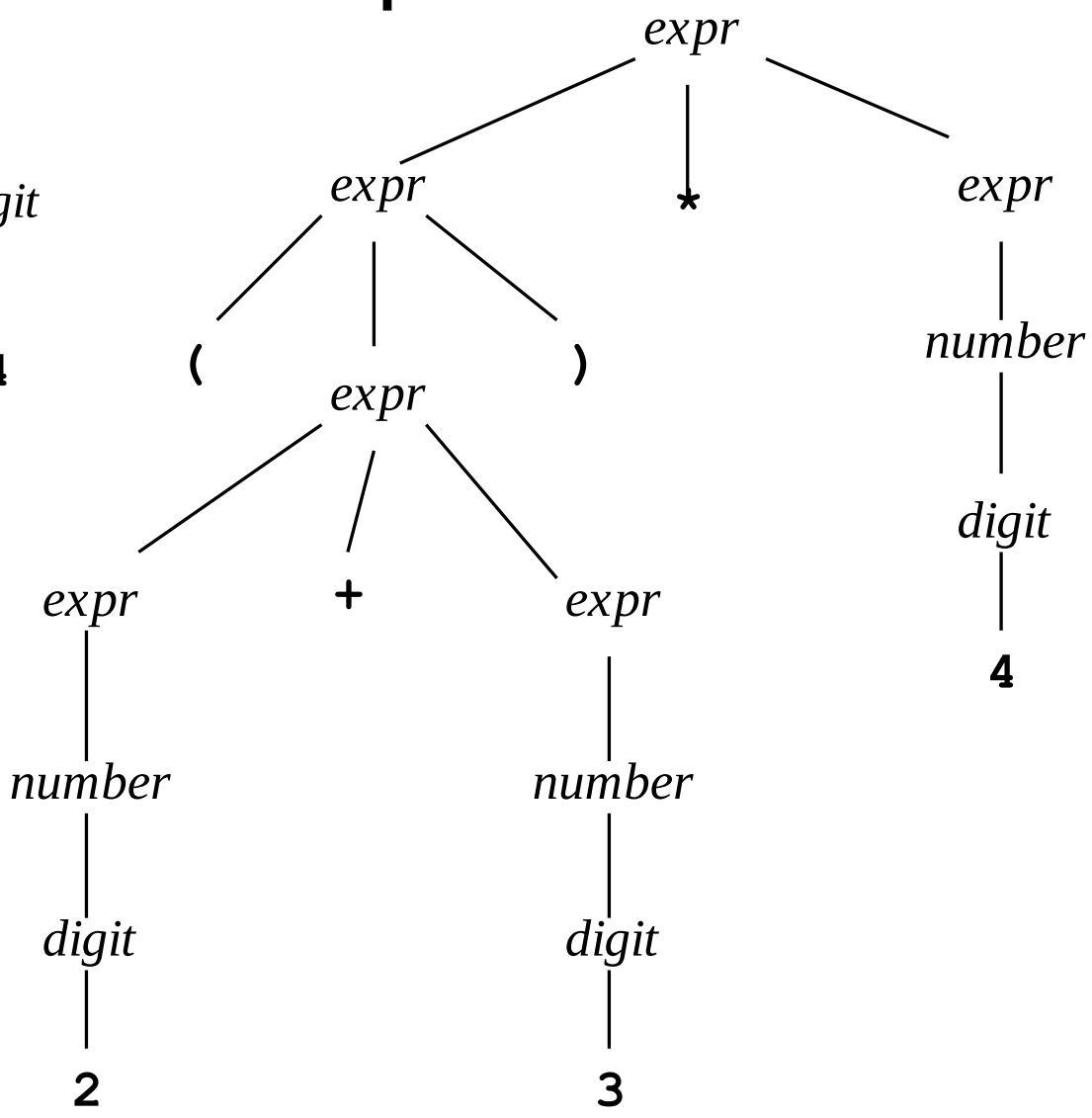
  $\Rightarrow$ *digit* `34` $\Rightarrow$ `234`

- **A parse tree better expresses the structure inherent in a derivation.**
- **Two sample parse trees on next slide.**

Example 1:

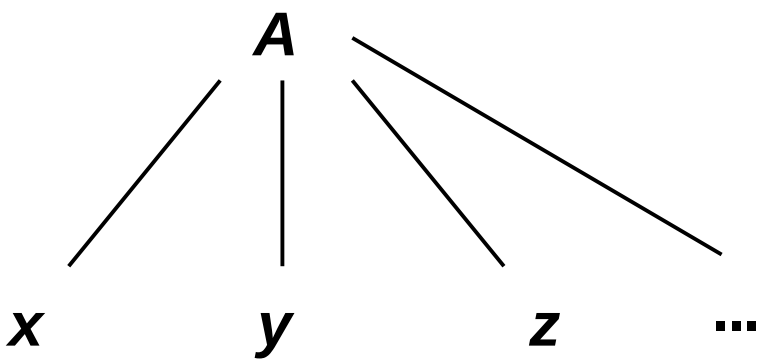


Example 2:



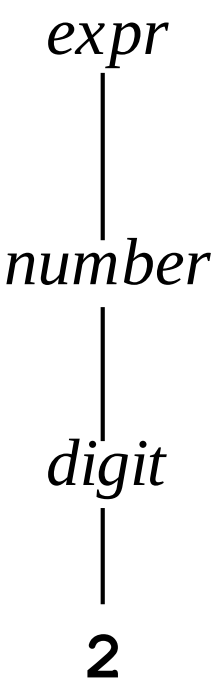
Notes on Parse Trees

- **Leaves are tokens, interior nodes are nonterminals, the root node is the start symbol.**
- **Every replacement in a derivation using a grammar rule *A* → *xyz…* corresponds to the creation of children at the node labeled *A*:**

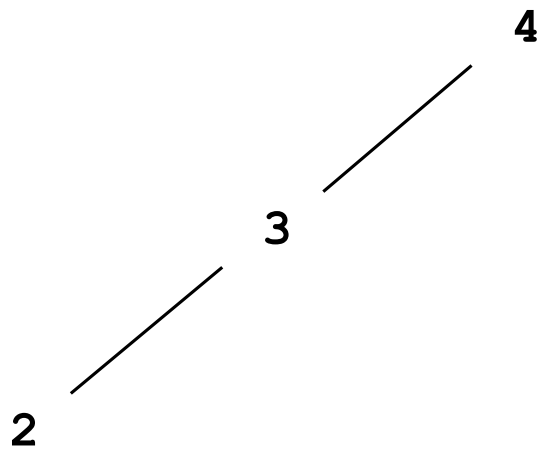


Abstract Syntax Trees

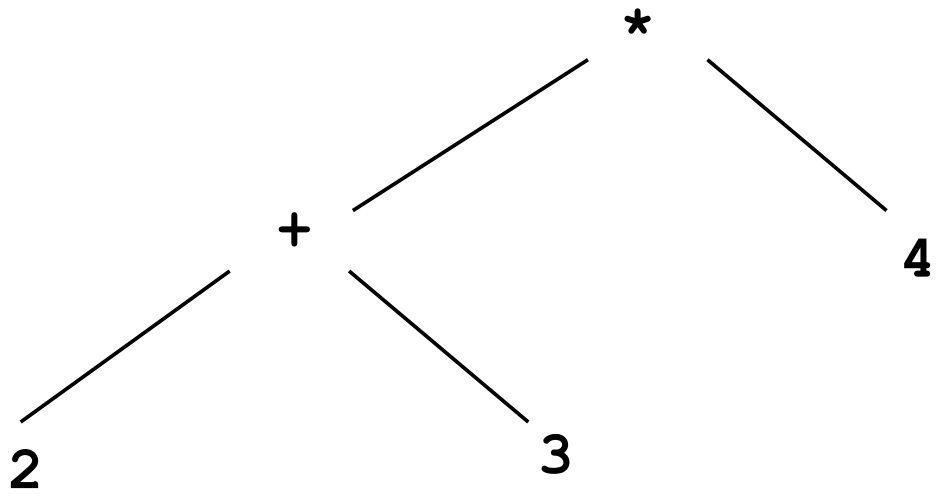
- **Parse trees are still too detailed in their structure, since every step in a derivation is expressed as nodes:**
- **We would really like this to be just a “2” node.**
- **An (abstract) syntax tree condenses a parse tree to its essential structure, removing such “cascades.”**



Example 1:



Example 2:



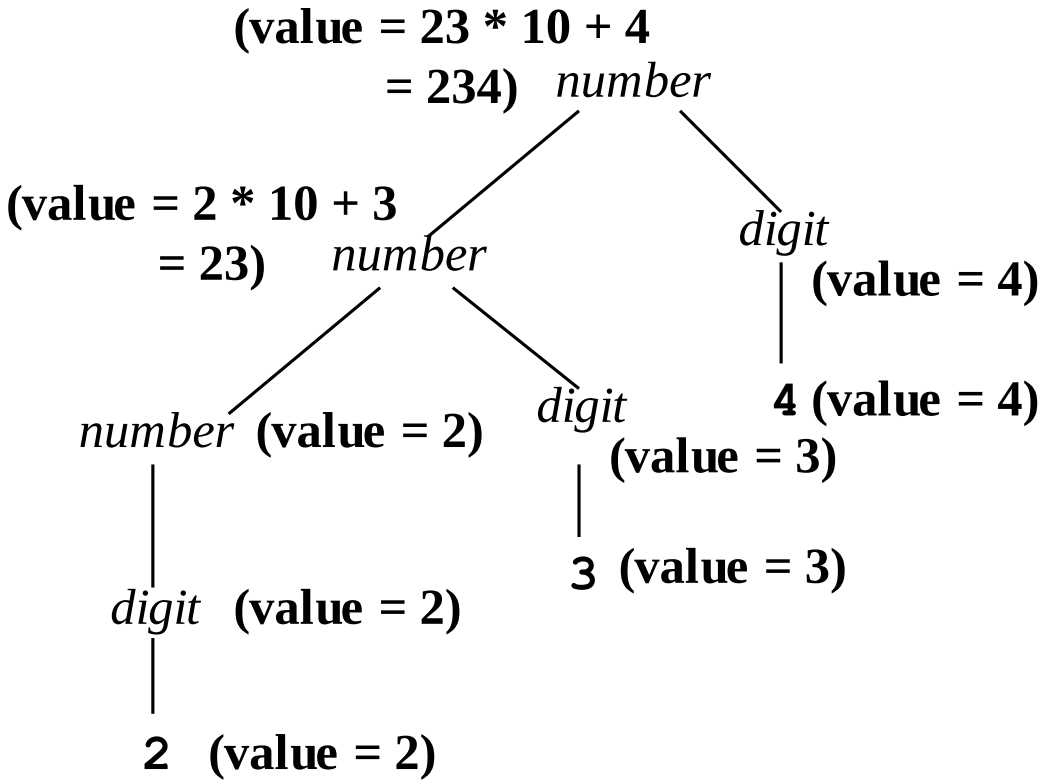
Note how much more compact -- even parentheses have disappeared!

Principle of syntax-directed semantics (or semantics-based syntax):

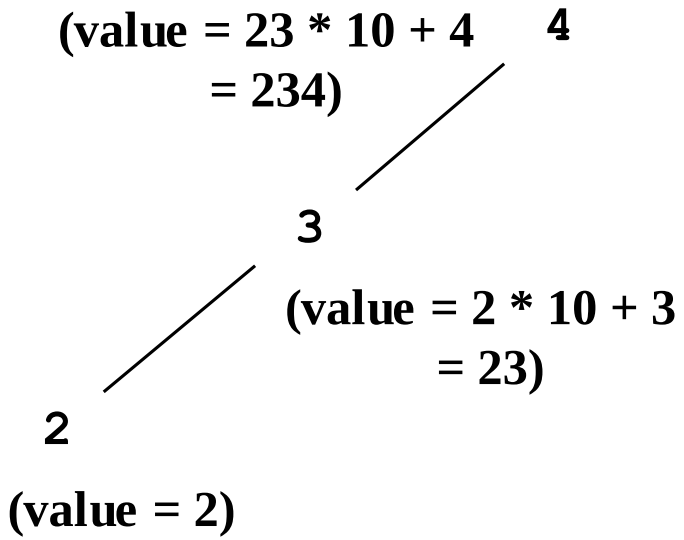
- **The parse tree and the abstract syntax tree must have a structure that corresponds to the computation to be performed.**

Example 1:

Parse tree

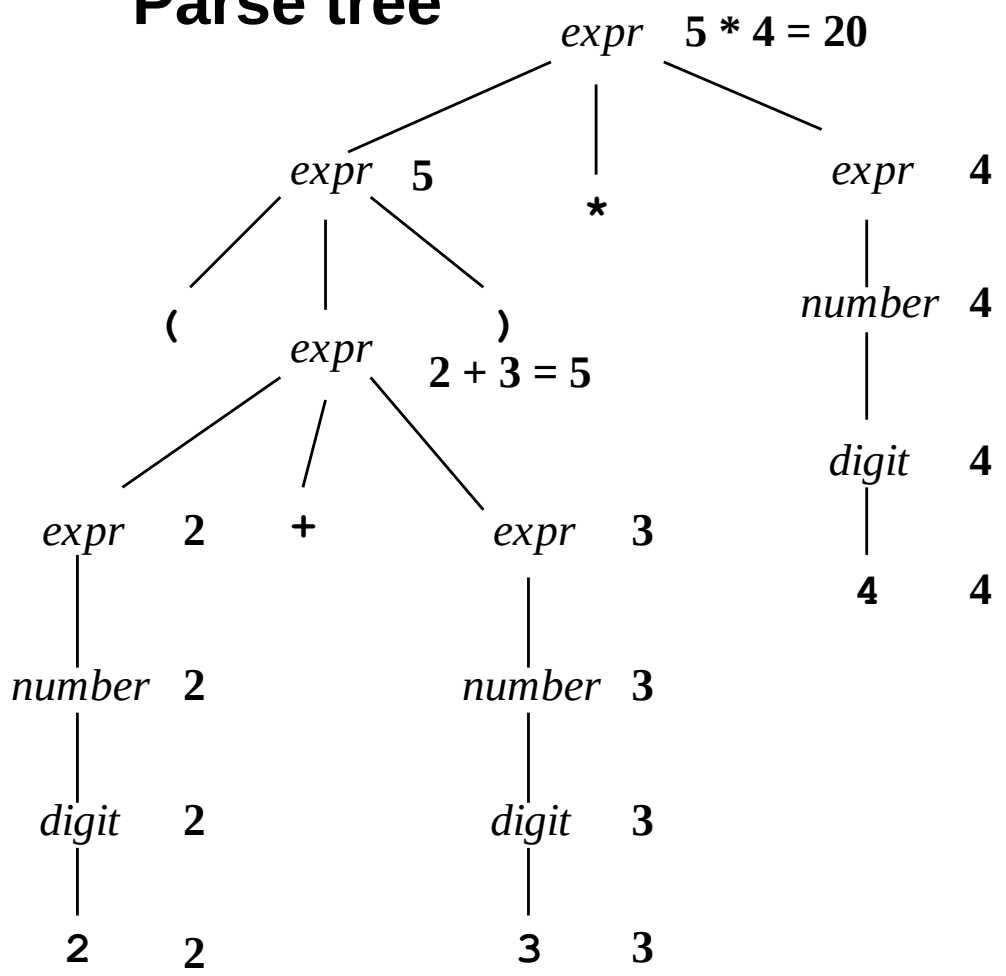


Abstract syntax tree

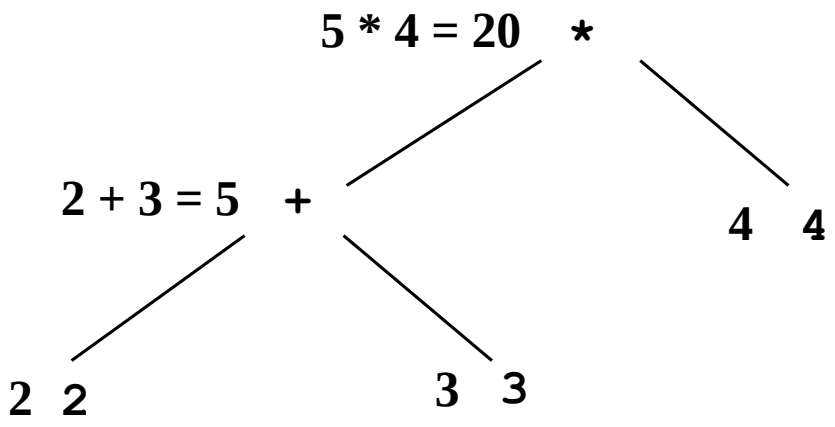


Example 2:

Parse tree



Abstract syntax tree



Ambiguity

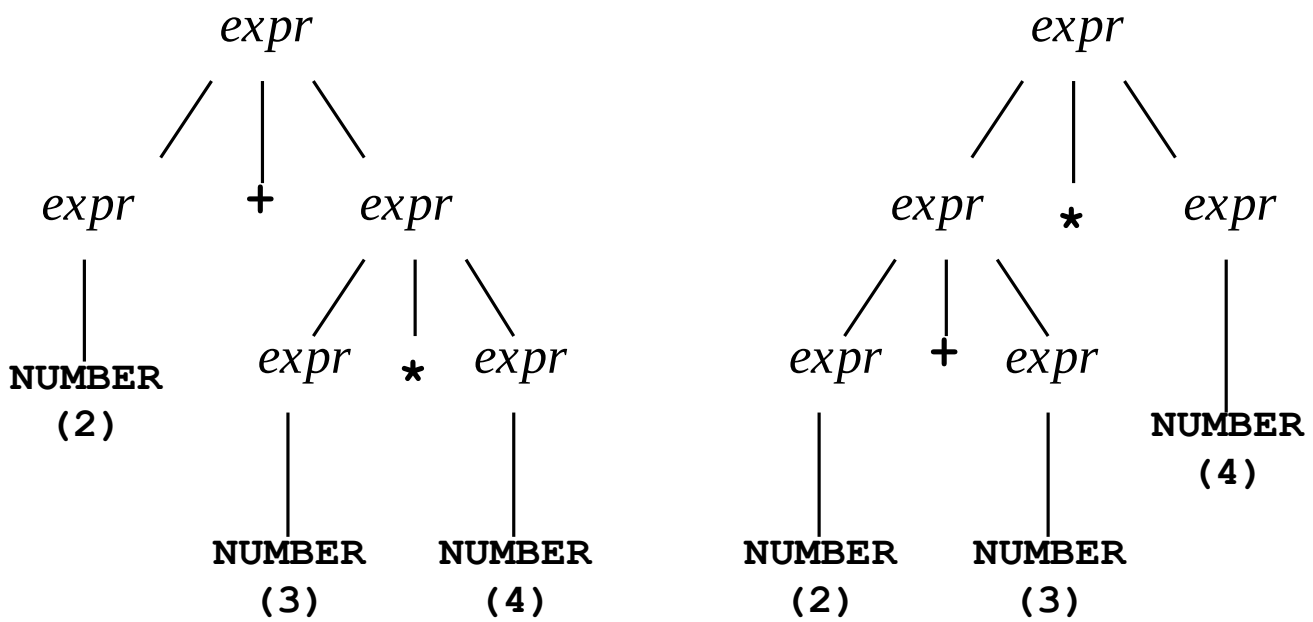
- **Grammars don't always specify unique parse trees for every string in the language: a grammar is ambiguous if some string has two distinct parse (or abstract syntax) trees (not just two distinct derivations).**
- **Ambiguity is usually bad and must be removed.**
- **Semantics help in determining which parse tree is correct.**
- **Often the grammar can be rewritten to make the correct choice.**

Example of Ambiguity

- **Grammar:**

  expr → *expr* **+** *expr* | *expr* ***** *expr*
  | **(** *expr* **)** | **NUMBER**

- **Expression: 2 + 3 * 4**
- **Parse trees:**

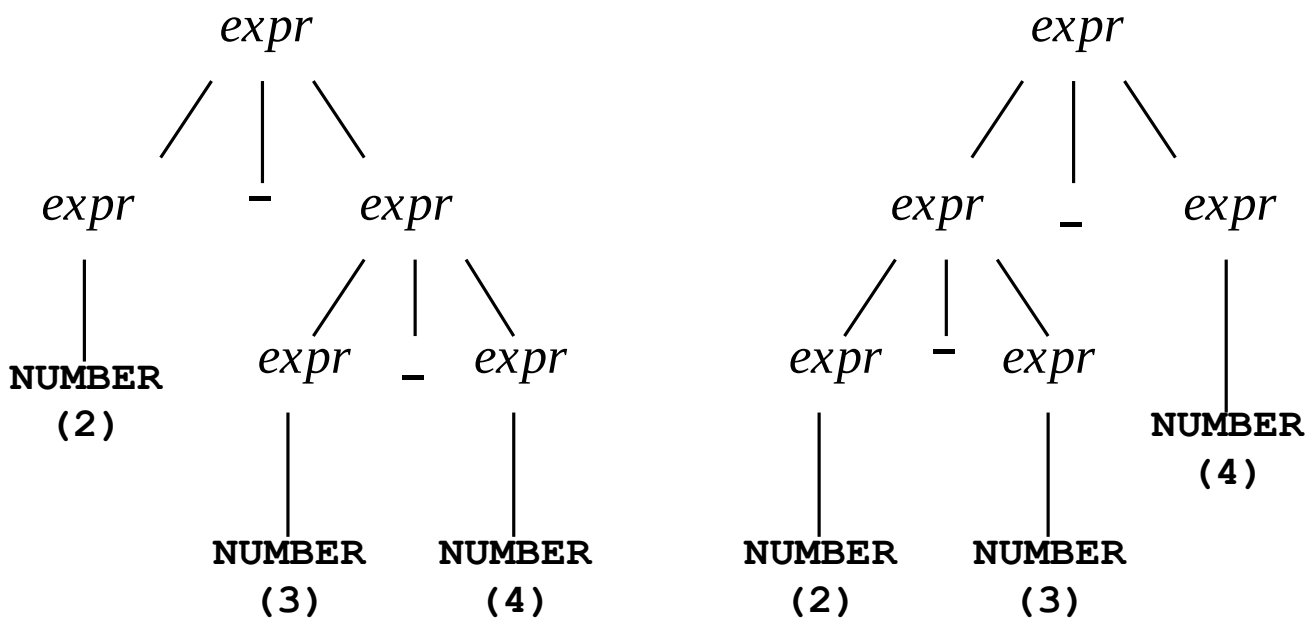


Another Example of Ambiguity

- **Grammar (with subtraction):**

  expr → *expr* **+** *expr* | *expr* **–** *expr*
  | **(** *expr* **)** | **NUMBER**

- **Expression: 2 – 3 – 4**
- **Parse trees:**



Resolving these ambiguities

- **The first example is a precedence issue, the second is an associativity issue.**
- **Use recursion to specify associativity, new rules (a "precedence cascade") to specify precedence:**

  expr → *expr* **+** *term* | *term*

  term → *term* ***** *factor* | *factor*

  factor→ **(** *expr* **)** | **`NUMBER`**

- **Note how left recursion expresses left associativity, and higher precedence means "lower" in the cascade.**

Assume the following rules of associativity and precedence for expressions

- **Precedence:**
  - **Highest** ***, /, not**
  - **+, -, &, mod**
  - **- (unary)**
  - **=, ≠, <, ≤, ≥, >**
  - **and**
  - **Lowest** **or, xor**
- **Associativity: left to right**
- **Write a CFG for the expression. Assume the only operands are the names a, b, c, d, and e.**

Recursion to specify associativity
Precedence cascade to specify precedence

<expr> ::= <expr> or <e1> | <expr> xor <e1> | <e1>

<e1> ::= <e1> and <e2> | <e2>

<e2> ::= <e2> = <e3> | <e2> /= <e3> | <e2> < <e3>
| <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>

<e3> ::= <e4> | -<e4>

<e4> ::= <e4> + <e5> | <e4> - <e5> | <e4> & <e5> |
<e4> mod <e5> | <e5>

<e5> ::= <e5> * <e6> | <e5> / <e6> | not <e5> | <e6>

<e6> ::= a | b | c | d | e | const | (<expr>)

Is there an alternative? Yes, but they both change the language:

- **Fully-parenthesized expressions:**
  expr → **(** *expr* **+** *expr* **)** | **(** *expr* ***** *expr* **)**
  | **NUMBER**
  so: ((2 + 3) * 4)
  and: (2 + (3 * 4))
- **Prefix expressions:**
  expr → **+** *expr expr* | ***** *expr expr*
  | **NUMBER**
  so: + + 2 3 4
  and: + 2 * 3 4

Scheme uses prefix form, but keeps the parentheses. Why?

- **Scheme allows any number of arguments to the arithmetic operators:**

  expr → **(***op exprlist* **)** | **NUMBER**

  exprlist → *exprlist expr* | *empty*

  empty →

  so: `(+)`, `(+ 1)`, `(+ 1 2 3 4)`, etc.

  [– and / require at least one argument]

Unary Minus (Ex. 4.13(a), p.116)

- **Add unary – (negation) so that at most one unary minus is allowed in each expression, and it must come at the beginning of an expression:**
  - **–2 + 3 is legal (and equals 1)**
  - **–2 + (–3) is legal**
  - **–2 + –3 is not legal**
- **Answer:**

  expr* → *expr* + *term* | *term* | – *term

Extended BNF Notation

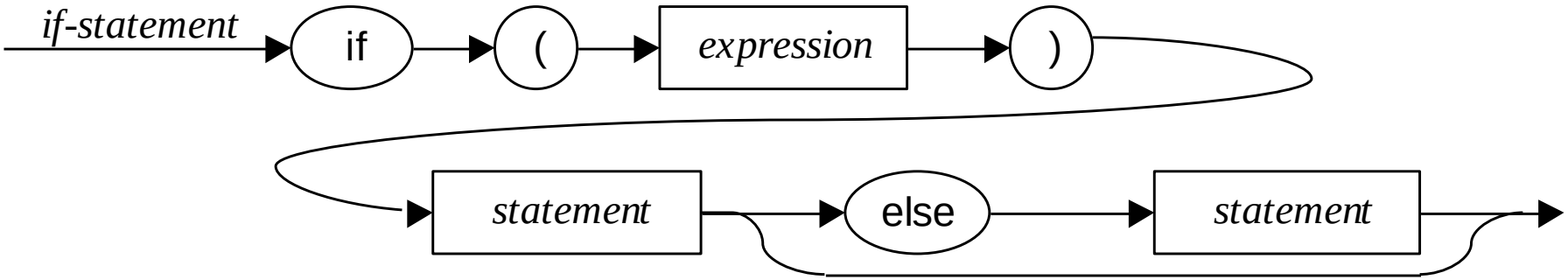
- **Notation for repetition and optional features.**
- **{…} expresses repetition:**
  expr* → *expr* + *term* | *term **becomes**
  ***expr* → *term* { + *term* }**
- **[…] expresses optional features:**
  if-stmt* → `if (` *expr* `)` *stmt
  | `if (` *expr* `)` *stmt* `else` *stmt*
  becomes
  ***if-stmt* → `if (` *expr* `)` *stmt* [`else` *stmt*]**

Notes on use of EBNF

- **Use {…} only for left recursive rules:**
  expr* → *term* + *expr* | *term
  should become *expr* → *term* [+ *expr*]
- **Do not start a rule with {…}: write**
  ***expr* → *term* { + *term* }, not**
  expr* → { *term* + } *term
- **Exception to previous rule: simple token repetition, e.g. *expr* → { – } *term* …**
- **Square brackets can be used anywhere, however:**
  expr* → *expr* + *term* | *term* | *unaryop term
  should be written as
  ***expr* → [*unaryop*] *term* { + *term* }**

Syntax Diagrams

- **An alternative to EBNF.**
- **Rarely seen any more: EBNF is much more compact.**
- **Example (if-statement, p. 101):**



Standard Grammar Categories

- **Declarations, sometimes called *definitions*. Java example:**

  ClassDeclaration: `class` *Identifier [* `extends` *Type]*
  [`implements` *TypeList] ClassBody*

- **Statements. Java example:**

  Statement: `throw` *Expression ;* | … *(other options)*

- **Expressions, such as the running expression grammar example in this chapter.**
- **Sequences of things (expressions, statements, declarations). Java example:**

  Block: `{` *BlockStatements* `}`

  BlockStatements: { BlockStatement }

Parsing

- **Only an overview here - see a compiler course: there are many algorithms and tools.**
- **Hand-written parsers almost always use an algorithm called recursive-descent that every computer professional should know.**
- **Based on EBNF, it models the grammar with a function for every nonterminal: the body of the function is given by the right-hand side of the grammar rule(s) for that nonterminal: tokens are matched from the input, and nonterminals are "called."**

BNF ->EBNF

- **expr -> expr + term | term**
- Two fatal difficulties arise if we were to write expr a recursive –descent procedure
  - Recursive call for the first rule
  - There is no way to decide which of the two choices to take until + is seen
  - Problem of left recursion
- EBNF is the solution
  - ***expr → term { + term }***

EBNF and Syntax diagrams

- **EBNF and Syntax Diagrams correspond naturally to the code of a recursive-descent parser**
- **Main reason for their use**

Recursive-descent example

- **Grammar:**
  ***expr* → *term* { + *term* }**
  ***term* → *factor* { ∗ *factor* }**
  factor* → (*expr*) | *number
- **Code sketch (see Fig. 4.12, pp. 105-107 for a complete example):**

```
expr()
{ term();
  while (token == '+')
  { match(token);
    term();
  }
}
```

```
factor()
{ if (token == '(')
  { match(token);
    expr();
    match(')');
  }
  else number();
}
```

The Parsing Problem

- **Making decisions on what to expect next in a parse based on the next input token can be tricky.**
- **In difficult situations, we must compute the First symbols of grammar rule choices: given $A \rightarrow \alpha_1 \mid \alpha_2$ we can decide which α_i to use only if First(α_1) $\cap$ First(α_2) is empty.**
- **Example: First(*expr*) = { `(` `NUMBER` }**

First condition that predictive parsing requires

- **First sets of any two choices must not have any tokens in common**
  - Example: factor -> (expr) | number
  - This condition for predictive parsing is satisfied, since
  - First (expr) ∩ first(number) = {(} ∩ {0,..,9}
  - = Ø

Second condition for predictive parsing requires

- When structures are optional
  - Example: S ->B [A] D
- if *A* is optional, then First(*A*) ∩ Follow(*A*) should be empty.
- Computation of first and follow sets can also be useful in recursive-descent parsing tests

The Parsing Problem (cont.)

- **Additional problem: trying to decide whether an optional construct is actually present.**
- **May have to compute Follow sets of nonterminals: if *A* is optional, then First(*A*) ∩ Follow(*A*) should be empty.**
- **Example: Exercise 4.49, p. 121, where *list* is optional:** *list* → *expr* [*list*] **. (See solution in posted answers.)**

Scanning/parsing tools

- **Scanners and parsers can be automatically generated from regular expressions and grammars.**
- **Unix tools Lex (Lesk, 1975) and Yacc (Johnson, 1975) are the most popular.**
- **Modern free versions are Gnu Bison and Flex ("Fast lex").**
- **Compiler courses cover these.**
- **Context-free grammars that meet the requirements for Yacc are called LALR(1) grammars (see a compiler course).**

Lexics vs. Syntax vs. Semantics

- **Division between lexical and syntactic structure is not fixed: a number can be a token or defined by a grammar rule.**
- **Implementation can often decide (scanners are faster, but parsers are more flexible).**
- **Division between syntax and semantics is in some ways similar: some authors call all static structure "syntax".**
- **Our view: if it isn't in the grammar (or the disambiguating rules), it's semantics.**