1 Syntax

- Informal *vs* formal specification
- Regular expressions
- Backus Naur Form (BNF)
- Extended Backus Naur Form (EBNF)
- Case study: Calc syntax

What is syntax?

- The **syntax** of a PL is concerned with the *form* of programs: how expressions, commands, declarations, and other constructs are arranged to make a well-formed program.
- When learning a new PL, we need to learn the PL's syntax.
- The PL's syntax must be specified. Examples alone do not show the PL's generality:

```
if n > 0 : write( n )
```

What is allowed here?
– a simple command?
– a sequence of commands?

What is allowed here?
– a variable?
– an arbitrary expression?

University of Glasgow

Informal *vs* formal specification

- An **informal specification** is one expressed in natural language (such as English).
- A **formal specification** is one expressed in a precise notation.
- Pros and cons of formal specification:
  - \+ more precise
  - \+ usually more concise
  - \+ less likely to be ambiguous, inconsistent, or incomplete
  - – accessible only to those familiar with the notation.

Example: informal *vs* formal syntax

- *Informal* syntax of some commands in a C-like language:

  A while-command consists of ‘`while`’, followed by an expression enclosed in parentheses, followed by a command.

  A sequential-command consists of a sequence of one or more commands, enclosed by ‘`{`’ and ‘`}`’.

- *Formal* syntax (using EBNF notation):

  while-command = ‘`while`’ ‘`(`’ *expression* ‘`)`’ *command*

  sequential-command = ‘`{`’ *command* $^+$ ‘`}`’

Notations for formal specification of PL syntax

- Regular expressions (REs)
  - good for specifying syntax of lexical elements of programs (such as identifiers, literals, comments).
- Backus Naur Form (BNF)
  - good for specifying syntax of larger and nested program constructs (such as expressions, commands, declarations).
- Extended Backus Naur Form (EBNF)
  - combination of BNF and REs, good for nearly everything.

Running example: Calc

- **Calc** is a very simple calculator language, with:
  - variables named ‘`a`’, …, ‘`z`’
  - expressions consisting of variables, numerals, and arithmetic operators
  - assignment and output commands.
- Example Calc program:

```
set x = 13
set y = x*(x+1)
put x
put y/2
```

Regular expressions

- A **regular expression** (**RE**) is a kind of pattern.
- Each RE **matches** a set of strings
  - possibly an infinite set of strings.
- We can use REs for a variety of applications:
  - specifying a pattern of strings to be searched for in a text
  - specifying a pattern of filenames to be searched for in a file system
  - specifying the syntax of a PL’s lexical elements.

Example: REs

- Examples:

‘M’(‘r’|‘rs’|‘iss’)
- means ‘M’ followed by either ‘r’ or ‘rs’ or ‘iss’
- matches ‘Mr’, ‘Mrs’, ‘Miss’.

‘b’(‘an’)*‘a’
- means ‘b’ followed by zero or more occurrences of ‘an’ followed by ‘a’
- matches ‘ba’, ‘bana’, ‘banana’, etc.

(‘x’|‘abc’)*
- means zero or more occurrences of ‘x’ or ‘abc’
- matches ‘’, ‘x’, ‘abc’, ‘xx’, ‘xabc’, ‘abcx’, ‘abcabc’, ‘xxx’, ‘xxabc’, ‘xabcx’, ‘abcxx’, etc.

RE notation *(1)*

- Basic RE notation:
  - '*xyz*' — matches the string '*xyz*'
  - $RE_1 \mid RE_2$ — matches any string matched by *either* RE_1 *or* RE_2
  - $RE_1\ RE_2$ — matches any string matched by RE_1 *concatenated* with any string matched by RE_2
  - *RE* * — matches the concatenation of *zero or more* strings, each of which is matched by *RE*
  - (*RE*) — matches any string matched by *RE* (parentheses used for grouping)

RE notation *(2)*

- Additional RE notation:
  - *RE* $^?$ matches *either* the empty string *or* any string matched by *RE*
  - *RE* $^+$ matches the concatenation of *one or more* strings, each of which is matched by *RE*
- These additional forms are useful but not essential. They can be expanded into basic RE notation:

$RE^? = RE \mid$ ‘’

$RE^+ = RE\ RE^*$

Example: Calc lexicon *(1)*

- A Calc identifier consists of a single lower-case letter.
- The syntax of such identifiers is specified by the RE:

```
‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ |
‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ |
‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’
```

Example: Calc lexicon *(2)*

- A Calc numeral consists of one or more decimal digits. E.g.:

  `5   13   2000000000`

- The syntax of such numbers is specified by the RE:

  (‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’)$^+$

Example: alphanumeric identifiers

- Consider a PL in which an identifier consists of a sequence of one or more upper-case letters and digits, starting with a letter. E.g.:

```
X   A1   P2P   SOS
```

- The syntax of such identifiers is specified by RE:

```
('A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z')
('A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
 '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')*
```

one letter (the first parenthesised group)

zero or more letters and digits (the second parenthesised group)

Application of REs: Unix shell *(1)*

- The Unix shell scripting language uses an *ad hoc* pattern-matching notation in which:
  - `[…]` matches any one of the enclosed characters
  - `?` (on its own) matches any single character
  - `*` (on its own) matches any string of 0 or more characters.
- This a *restricted* variant of RE notation. (It lacks "$RE_1 | RE_2$" and "RE *".)

Application of REs: Unix shell *(2)*

- Example commands:

```
print bat.[chp]
```

prints files whose names are ‘bat.c’, ‘bat.h’, or ‘bat.p’

```
print bat.?
```

prints all files whose names are ‘bat.’ followed by any single character

```
print *.c
```

prints all files whose names end with ‘.c’

Application of REs: `egrep` *(1)*

- The Unix utility `egrep` uses the full pattern-matching notation, in which the following have their usual meanings:
  - $RE_1 | RE_2$
  - *RE*`*`
  - *RE*`+`
  - *RE*`?`
- It also provides extensions such as:
  - `[…]` matches any one of the enclosed characters
  - `.` matches any single character.

Application of REs: `egrep` *(2)*

- Example commands:

  `egrep "b[aei]t"` *file*

  finds all lines in *file* containing ‘`bat`’, ‘`bet`’, or ‘`bit`’

  `egrep "b.t"` *file*

  finds all lines in *file* containing ‘`b`’ followed by any character followed by ‘`t`’.

  `egrep "b(an)*a"` *file*

  finds all lines in *file* containing ‘`b`’ followed by 0 or more occurrences of ‘`an`’ followed by ‘`a`’.

Application of REs: Java pattern matching

- Some Java classes also use the full pattern-matching notation, with the same extensions as `egrep`:
  - `[…]` matches any one of the enclosed characters
  - `.` matches any single character.
- Example code:

```
String s = …;
if (s.matches("b.t")) …
if (s.matches("b[aeiou]t")) …
if (s.matches("M(r|rs|iss)")) …
if (s.matches("b(an)*a")) …
```

Limitations of REs

- REs are not powerful enough to express the syntax of nested (embedded) phrases.
- In every PL, expressions can be nested:

```
n * (n + 1)
```

- In nearly every PL, commands can be nested:

```
while (r>0)
  { m = n; n = r;
    r = m-(n*(m/n)); }
```

Grammars

- To specify the syntax of nested phrases such as expressions and commands, we need a (*context-free*) *grammar*.
- The **grammar** of a language is a set of rules specifying how the phrases of that language are formed.
- Each rule specifies how each phrase may be formed from symbols (such as words and punctuation) and simpler phrases.

Example: mini-English grammar *(1)*

- **Mini-English** consists of simple sentences like:

  `I smell a rat .`

  `the cat sees me .`

- The following symbols occur in mini-English sentences:

  `'a'` `'cat'` `'I'` `'mat'` `'me'` `'rat'`
  `'see'` `'sees'` `'smell'` `'smells'` `'the'` `'.'`

  terminal symbols

- The grammar uses the following symbols to denote mini-English phrases:

  sentence *subject* *object* *noun* *verb*

  nonterminal symbols

Example: mini-English grammar *(2)*

- Production rules of the mini-English grammar:

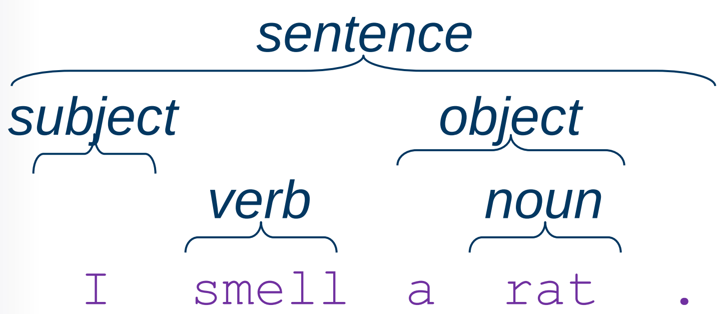
sentence	=	*subject verb object* ‘`.`’
subject	=	‘`I`’ | ‘`a`’ *noun* | ‘`the`’ *noun*
object	=	‘`me`’ | ‘`a`’ *noun* | ‘`the`’ *noun*
noun	=	‘`cat`’ | ‘`mat`’ | ‘`rat`’
verb	=	‘`see`’ | ‘`sees`’ | ‘`smell`’ | ‘`smells`’

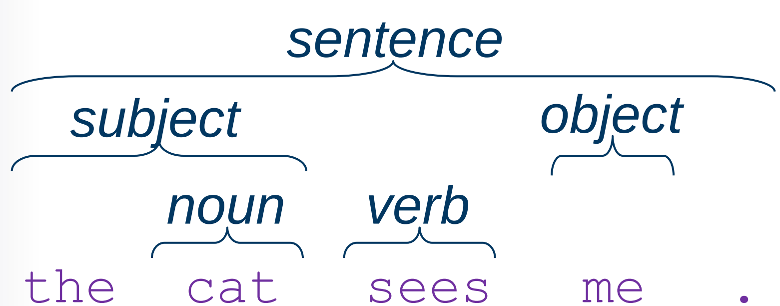
read as
“A *sentence* consists of
a *subject* followed by
a *verb* followed by
an *object* followed by ‘`.`’.”

read as
“A *subject* consists of the word ‘`I`’ alone,
or the word ‘`a`’ followed by a *noun*,
or the word ‘`the`’ followed by a *noun*.”

Example: mini-English grammar *(3)*

- How sentences are structured:





- The structure of a sentence can be shown by a syntax tree *(see later)*.

Grammars, symbols, production rules

- A **context-free grammar** (or just **grammar**) consists of:
  - a set of **terminal symbols**
  - a set of **nonterminal symbols**
  - a **sentence symbol**
  - a set of **production rules**.

Each terminal symbol is a symbol that may occur in a sentence.

Each nonterminal symbol stands for a phrase that may form part of a sentence.

The sentence symbol is the nonterminal symbol that stands for a complete sentence.

Each production rule specifies how phrases are composed from terminal symbols and sub-phrases.

BNF notation *(1)*

- **Backus Naur Form** (**BNF**) is a notation for expressing a grammar.
- A simple production rule in BNF looks like this:

  N = α

  α is a sequence of terminal and nonterminal symbols

  “=” is read as “consists of”

  N is a nonterminal symbol

- Example (mini-English):

  sentence = *subject* *verb* *object* ‘.’

BNF notation *(2)*

- More generally, a production rule in BNF may have several alternatives on its right-hand side:

  $N = \alpha \mid \beta \mid \gamma$

  each of α, β, γ is a sequence of terminal and nonterminal symbols

  "|" is read as "or".

- Example (mini-English):

  subject = '`I`' | '`a`' *noun* | '`the`' *noun*

Example: Calc grammar in BNF *(1)*

- Terminal symbols:

  `'put'  'set'`
  `'='  '+'  '-'  '*'  '('  ')'`
  `'\n'`
  `'a'  'b'  'c'  …  'z'  '0'  '1'  …  '9'`

- Nonterminal symbols:

  prog   *com*
  expr   *prim*
  num   *id*

- Sentence symbol:

  prog

Example: Calc grammar in BNF *(2)*

- Production rules:

```
prog  =  eof
      |  com prog

com   =  'put' expr eol
      |  'set' id '=' expr eol

expr  =  prim
      |  expr '+' prim
      |  expr '-' prim
      |  expr '*' prim

prim  =  num
      |  id
      |  '(' expr ')'
```

A *prog* consists of just an *eof*, or alternatively a *com* followed by a *prog*.

In other words, a *prog* consists of a sequence of zero or more *coms* followed by an *eof*.

Example: Calc grammar in BNF *(3)*

- Production rules *(continued)*:

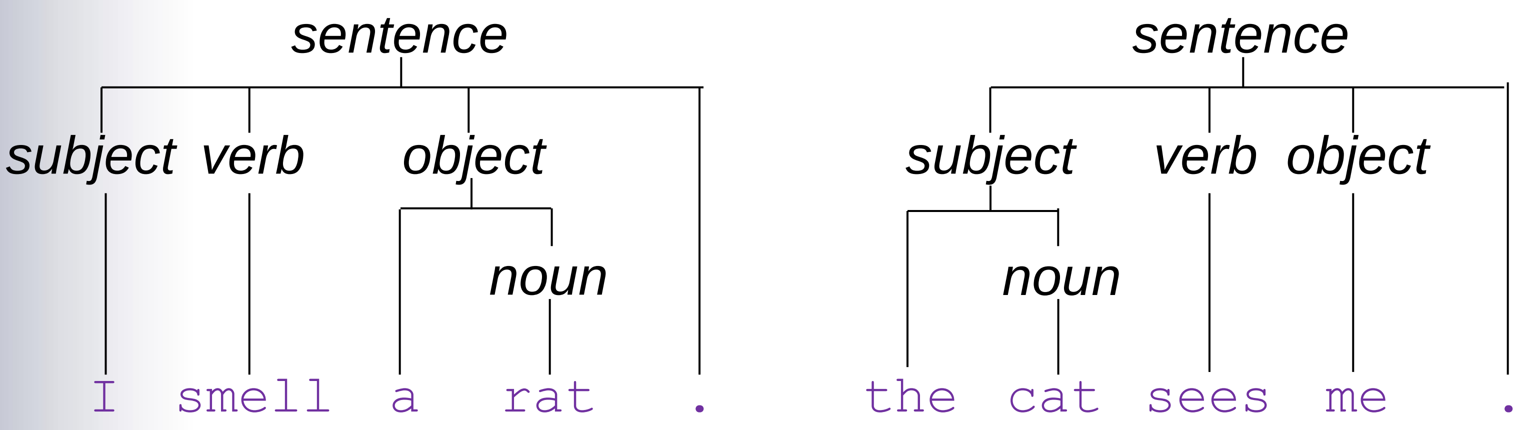
num	=	*digit* | *num digit*
id	=	*letter*
letter	=	‘`a`’ | ‘`b`’ | ‘`c`’ | … | ‘`z`’
digit	=	‘`0`’ | ‘`1`’ | … | ‘`9`’
eol	=	‘`\n`’

Phrase structure

- A grammar defines how phrases may be formed from sub-phrases in the language. This is called **phrase structure**.
- Every phrase in the language has a *syntax tree* that explicitly represents its phrase structure.

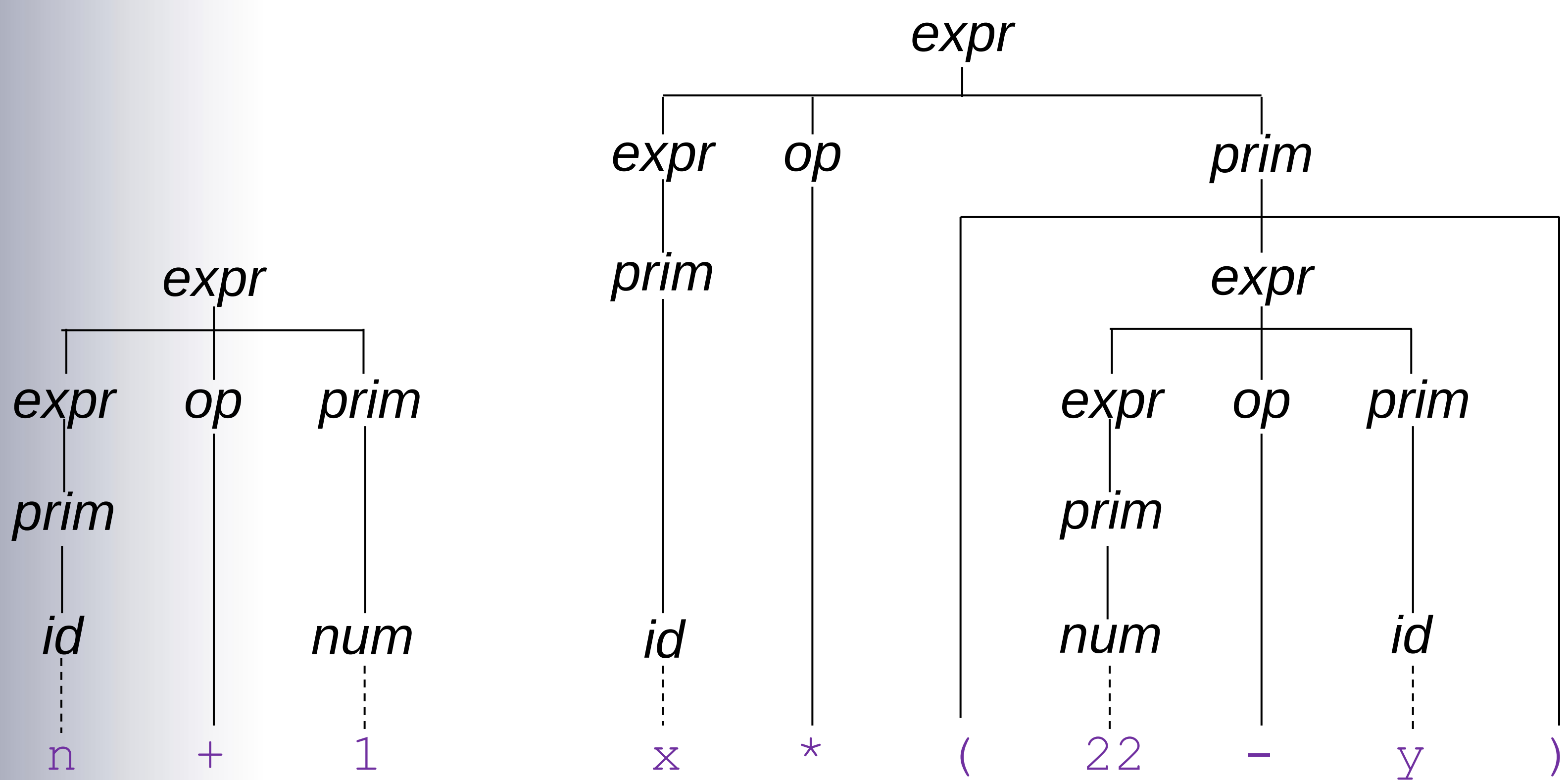
Example: mini-English syntax trees

- Syntax trees of mini-English sentences:



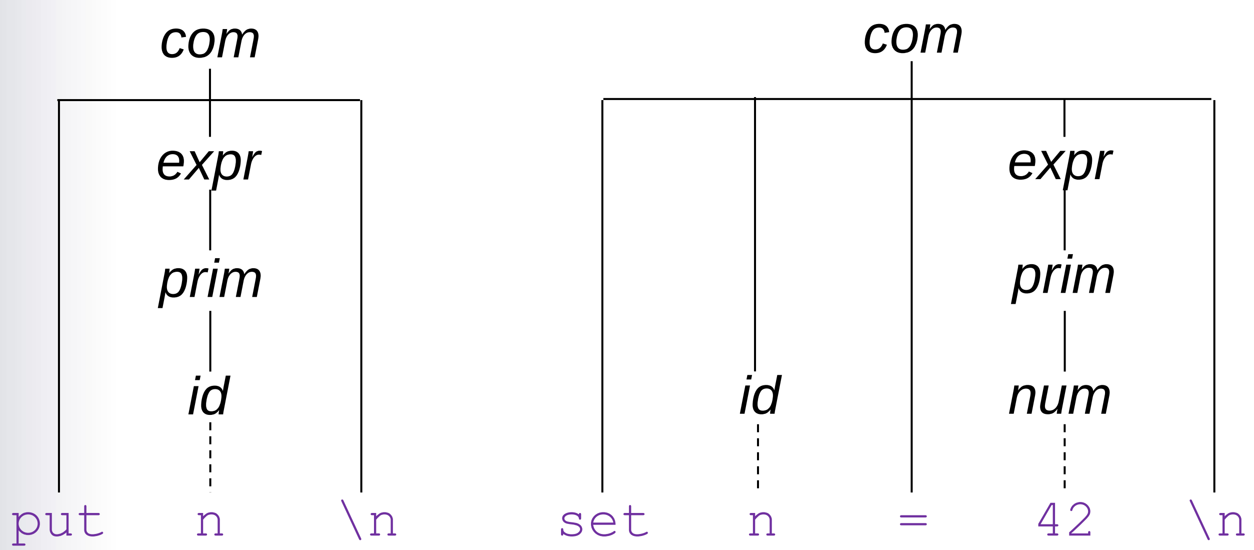
Example: Calc syntax trees *(1)*

- Syntax trees of Calc expressions:



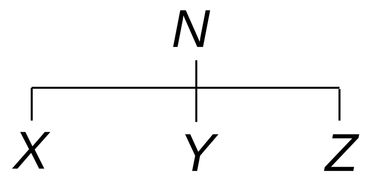
Example: Calc syntax trees *(2)*

- Syntax trees of Calc commands:



Syntax trees

- Consider a grammar *G*.
- A **syntax tree of *G*** is a tree with the following properties:
  - Every *terminal* node is labeled by a *terminal* symbol of *G*.
  - Every *nonterminal* node is labeled by a *nonterminal* symbol of *G*.
  - A nonterminal node labeled *N* may have children labeled *X*, *Y*, *Z* (from left to right) only if *G* has a production rule *N* = *X Y Z* or *N* = … | *X Y Z* | …



Phrases

- If *N* is a nonterminal symbol of *G, a* **phrase** of class *N* is a string of terminal symbols labeling the terminal nodes of a syntax tree whose root node is labeled *N*.
  - *Note:* The terminal nodes must be visited from left to right.
- E.g., phrases in Calc:
  - ‘`x*(22-y)`’ is a phrase of class *expr*
  - ‘`set n = 42 \n`’ is a phrase of class *com*
  - ‘`set n = 42 \n put x*(22-y) \n`’ is a phrase of class *prog*.

Sentences and languages

- If *S* is the sentence symbol of *G*, a **sentence** of *G* is a phrase of class *S*. E.g.:
  - ‘`set n = 42 \n put x*(22-y) \n`’ is a sentence of Calc.
- The **language** generated by *G* is the set of all sentences of *G*.
- Note: The language generated by *G* is typically infinite (although *G* itself is finite).

University of Glasgow

Phrase structure and semantics

- The above definition of a language is narrowly syntactic: a set of sentences.
- We are also interested in the language's *semantics* (i.e., the meaning of each sentence).
- A grammar does more than generate a set of sentences: it also imposes a phrase structure on each sentence (embodied in the sentence's syntax tree).
- Once we know a sentence's phrase structure, we can use it to ascribe a meaning to that sentence.

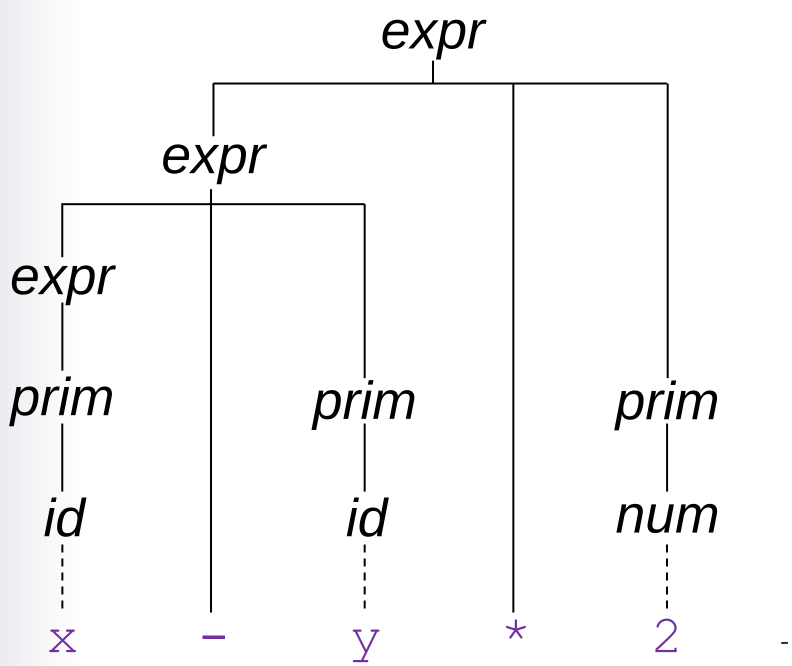
Example: expression structure *(1)*

- Consider this grammar (similar to Calc):

expr	=	*prim*
	|	*expr* ‘+’ *prim*
	|	*expr* ‘–’ *prim*
	|	*expr* ‘+’ *prim*
prim	=	*num*
	|	*id*
	|	‘(’ *expr* ‘)’

Example: expression structure *(2)*

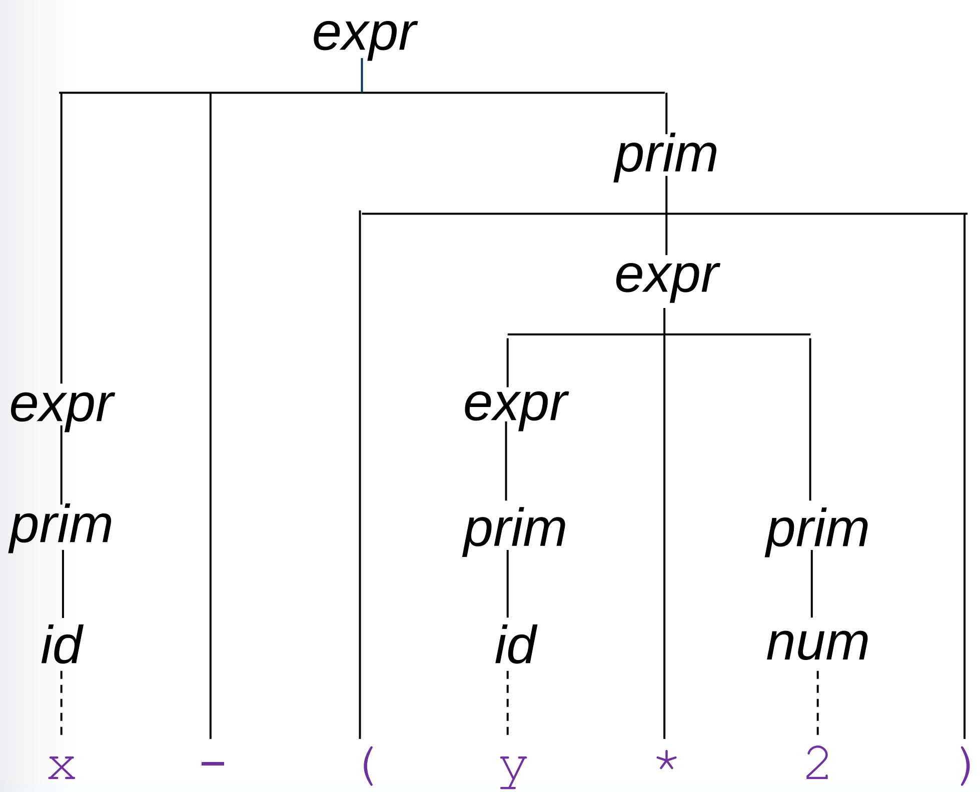
- In this grammar, operators ‘+’, ‘–’, and ‘*’ all have the same precedence . E.g.:



x–y*2 will be evaluated as (x–y)*2

Example: expression structure *(3)*

- But note that parentheses can always be used to control the evaluation:



Example: expression structure *(4)*

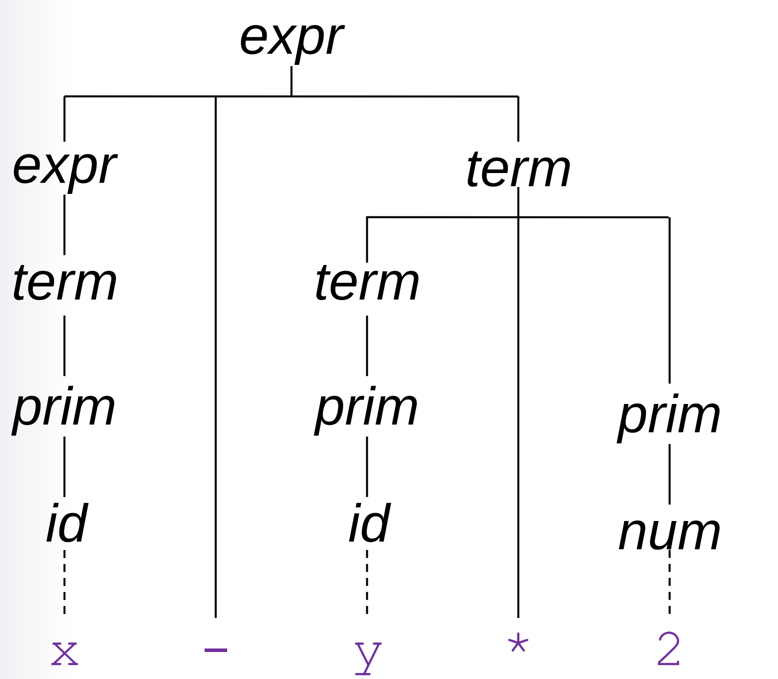
- Consider this different grammar:

expr	=	*term*
	|	*expr* '+' *term*
	|	*expr* '–' *term*
term	=	*prim*
	|	*term* '*' *prim*
prim	=	*num*
	|	*id*
	|	'(' *expr* ')'

- This grammar is typical of most PLs such as C and Java. It leads to a different phrase structure.

Example: expression structure *(5)*

- In this grammar, operator ‘`*`’ has higher precedence than ‘`+`’ and ‘`–`’. E.g.:



`x–y*2` will be evaluated as `x–(y*2)`

Ambiguity

- A phrase is **ambiguous** if it has more than one syntax tree.
- A grammar is **ambiguous** if any of its phrases is ambiguous.
- Ambiguity is common in natural languages such as English:
  - The peasants are revolting.
  - Time flies like an arrow. Fruit flies like a banana.
- The grammar of a PL should be unambiguous, otherwise the meaning of some programs would be uncertain.

Example: dangling "else" ambiguity *(1)*

- Part of the grammar of a fictional PL:

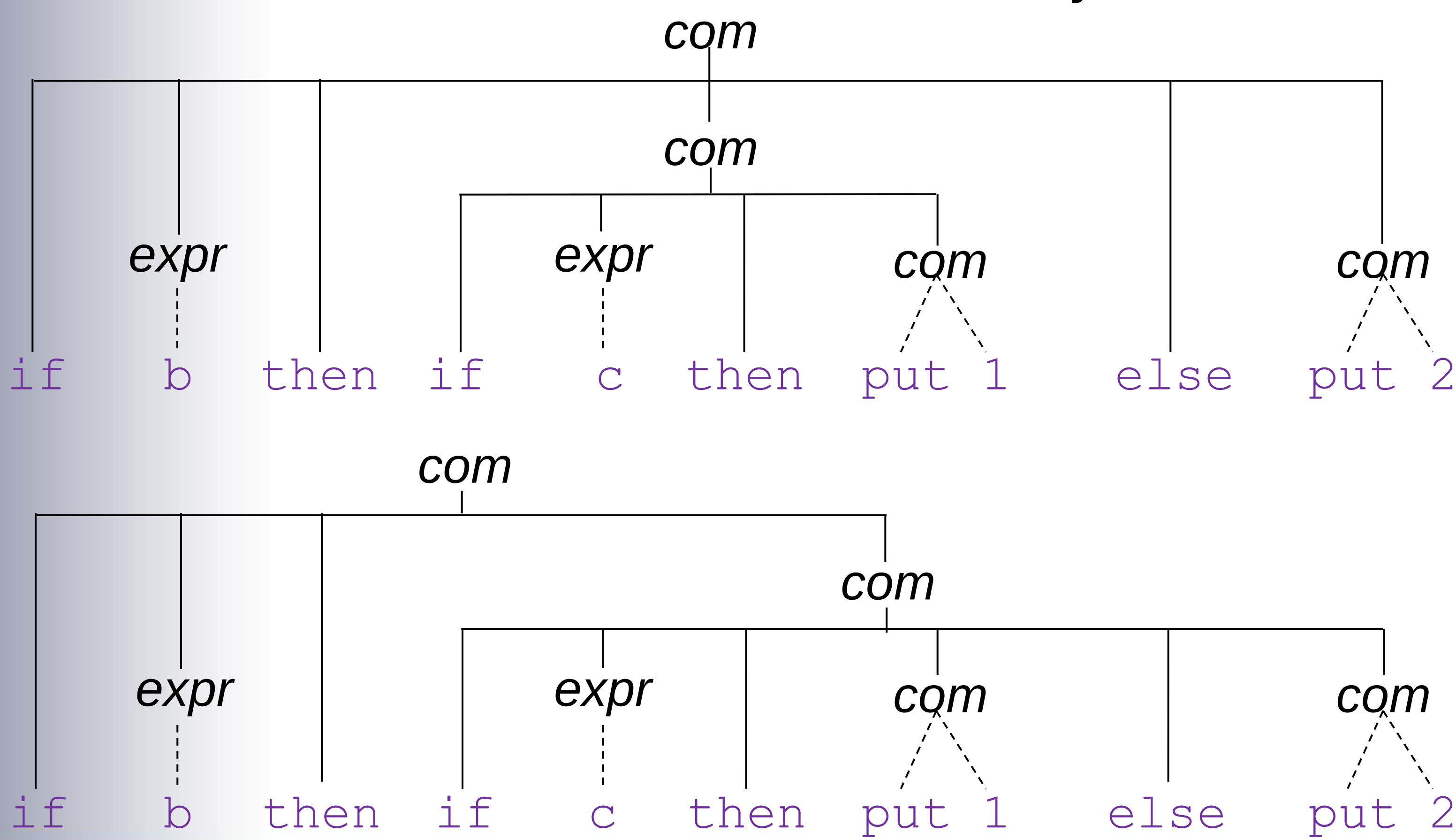
  com = '`put`' *expr*
  | '`if`' *expr* '`then`' *com*
  | '`if`' *expr* '`then`' *com* '`else`' *com*
  | …

- This makes some if-commands ambiguous, such as:

```
if b then if c then put 1 else put 2
```

Example: dangling "else" ambiguity *(2)*

- The above if-command has two syntax trees:



ENBF notation

- **Extended Backus Naur Form** (**EBNF**) is a combination of BNF and RE notation.
- An EBNF production rule has the form:

  N = *RE*

  where *RE* is a regular expression, expressed in terms of both terminal *and nonterminal* symbols.
- Example:

  sequential-command = '{' *command*$^{+}$ '}'
- EBNF is convenient for specifying all aspects of syntax.

Example: Calc syntax in EBNF *(1)*

- Production rules:

prog	=	*com* * *eof*
com	=	‘`put`’ *expr* *eol*
	|	‘`set`’ *id* ‘`=`’ *expr* *eol*
expr	=	*prim* (‘`+`’ *prim* | ‘`-`’ *prim* | ‘`*`’ *prim*)*
prim	=	*num*
	|	*id*
	|	‘`(`’ *expr* ‘`)`’

Example: Calc syntax in EBNF *(2)*

- Production rules *(continued)*:

id	=	‘`a`’ | ‘`b`’ | ‘`c`’ | … | ‘`z`’
num	=	(‘`0`’ | ‘`1`’ | … | ‘`9`’)$^+$
eol	=	‘`\n`’