Comp 204: Computer Systems and Their Implementation

Lecture 20: Parsing & Syntax

Today

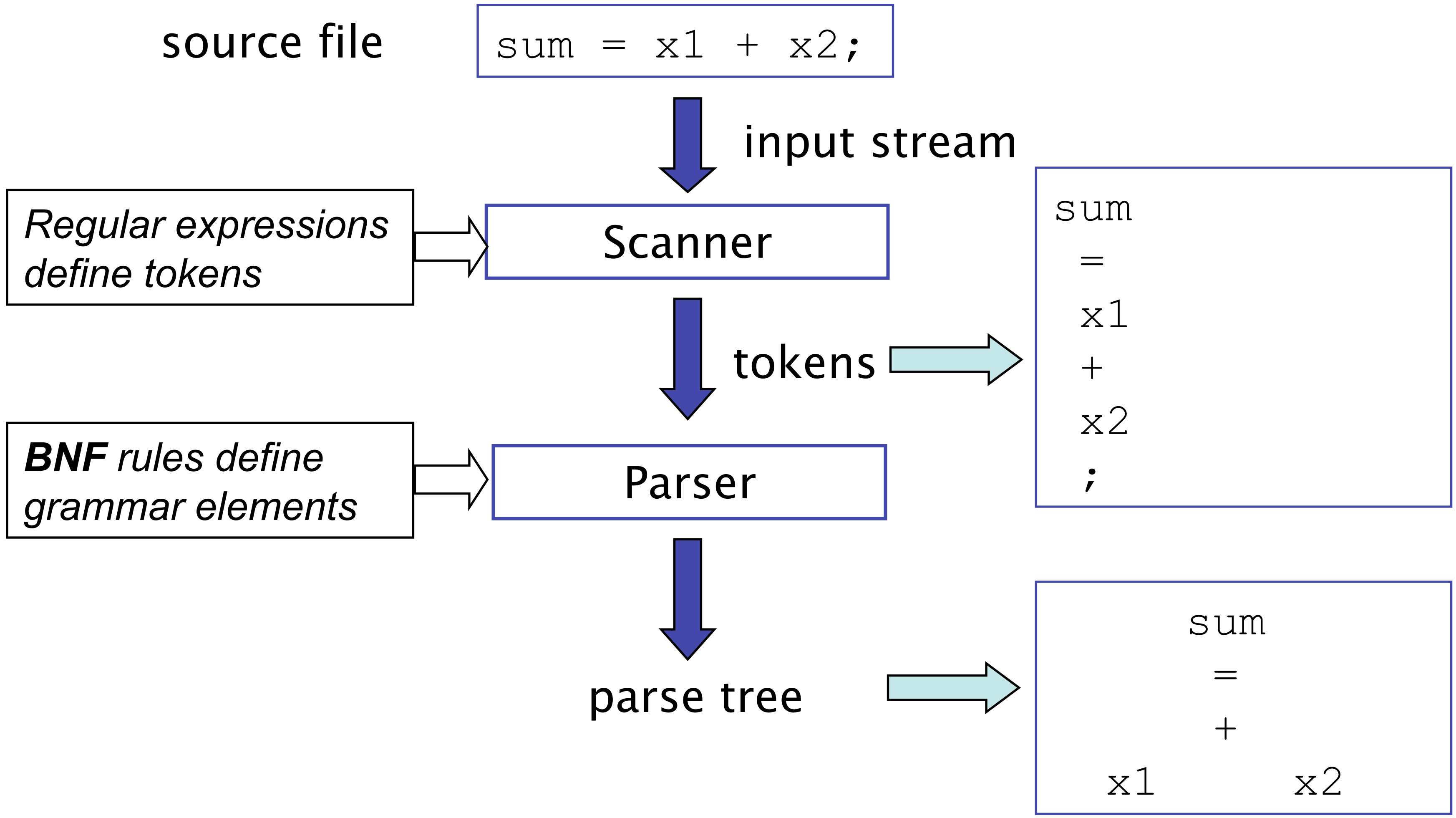
- Parsing
  - Context-free grammar & BNF
  - Example: The Micro language
  - Parse Tree
  - Abstract syntax tree

Parser (Syntax Analyser)

- Reads tokens and groups them into units as specified by language grammar
  i.e. it recognises syntactic phrases

- Parser must produce good errors and be able to recover from errors

Scanning and Parsing



Syntax

- Defines the structure of legal statements in the language
- Usually specified formally using a context-free grammar (CFG)
- Notation most widely used is Backus-Naur Form (BNF), or extended BNF
- A CFG is written as a set of rules (productions)
- In extended BNF:
  - {...} means zero or many
  - [...] means zero or one

Backus Naur Form

- **Backus Naur Form** (BNF) iw a standard notation for expressing ***syntax*** as a set of grammar rules.
  - BNF was developed by Noam Chomsky, John Backus, and Peter Naur.
  - First used to describe Algol.
- BNF can describe any ***context-free grammar***.
  - Fortunately, computer languages are mostly context-free.
- Computer languages remove *non-context-free* meaning by either
  - (a) defining more grammar rules or
  - (b) pushing the problem off to the semantic analysis phase.

A Context-Free Grammar

A grammar is *context-free* if all the syntax rules apply regardless of the symbols before or after (the context).

Example:

(1) *sentence* => *noun-phrase* *verb-phrase* .

(2) *noun-phrase* => *article* *noun*

(3) *article* => `a` | `the`

(4) *noun* => `boy` | `girl` | `cat` | `dog`

(5) *verb-phrase* => *verb* *noun-phrase*

(6) *verb* => `sees` | `pets` | `bites`

Terminal symbols:

'a' 'the' 'boy' 'girl' 'sees' 'pets' 'bites'

A Context-Free Grammar

A sentence that matches the *productions* (1) - (6) is valid.

a girl sees a boy
a girl sees a girl
a girl sees the dog
the dog pets the girl
a boy bites the dog
a dog pets the boy
...

To eliminate unwanted sentences without imposing *context sensitive* grammar, specify semantic rules:

"a boy may not bite a dog"

Backus Naur Form

- *Grammar Rules or Productions:* define symbols.

assignment_stmt **::=** *id* = *expression* ;

The *nonterminal symbol being defined.*

The *definition* (production)

Nonterminal Symbols: anything that is defined on the left-side of some production.

Terminal Symbols: things that are not defined by productions. They can be literals, symbols, and other *lexemes* of the language defined by lexical rules.

Identifiers: *id* ::= [A-Za-z_]\w*

Delimiters: ;

Operators: = + - * / %

Backus Naur Form (2)

- Different notations (same meaning):

  assignment_stmt ::= *id* = *expression* + *term*

  <*assignment-stmt*> => <*id*> = <*expr*> + <*term*>

  AssignmentStmt → *id* = *expression* + *term*

  ::=, =>, → mean "*consists* of" or "*defined as*"

- Alternatives (" | "):

```
expression  => expression + term
                | expression - term
                | term
```

- Concatenation:

```
number   => DIGIT number | DIGIT
```

Alternative Example

- The following BNF syntax is an example of how an arithmetic expression might be constructed in a simple language…

- Note the recursive nature of the rules

Syntax for Arithmetic Expr.

```
<expression> ::= <term> | <addop> <term> |<expression> <addop> <term>

<term> ::= <primary> | <term> <multop> <primary>

<primary> ::= <digit> | <letter> | ( <expression> )

<digit> ::= 0 | 1 | 2 |...| 9

<letter> ::= a | b | c |...| y | z

<multop> ::= * | /

<addop> ::= + | -
```

- Are the following expressions legal, according to this syntax?
  - i) -a
  - ii) b+c^(3/d)
  - iii) a*(c-(4+b))
  - iv) 5(9-e)/d

BNF rules can be recursive

```
expr   => expr + term
            | expr - term
            | term
term   => term * factor
            | term / factor
            | factor
factor => ( expr )  | ID | NUMBER
```

where the tokens are:

```
NUMBER := [0-9]+
ID     := [A-Za-z_][A-Za-z_0-9]*
```

Uses of Recursion

- **Repetition**

  expr => *expr* + *term*

  => *expr* + *term* + *term*

  => *expr* + *term* + *term* + *term*

  => *term* + ... + *term* + *term*

- Parser can recursively expand *expr* each time one is found
  - Could lead to arbitrary depth analysis
  - Greatly simplifies implementation

Example: The Micro Language

- To illustrate BNF parsing, consider an example imaginary language: the "*Micro*" language

1) A program is of the form

```
begin
        sequence of statements
end
```

2) Only statements allowed are

- assignment
- read (list of variables)
- write (list of expressions)

Micro

3) Variables are declared implicitly
   - their type is integer

4) Each statement ends in a semi-colon

5) Only operators are +, -
   - parentheses may be used

Micro CFG

1.	<program>	::=	begin <stat-list> end
2.	<stat-list>	::=	<statement> { <statement> }
3.	<statement>	::=	id := <expr> ;
4.	<statement>	::=	read (<id-list>) ;
5.	<statement>	::=	write (<expr-list>) ;
6.	<id-list>	::=	id { , id }
7.	<expr-list>	::=	<expr> { , <expr> }
8.	<expr>	::=	<primary> { <addop> <primary> }
9.	<primary>	::=	(<expr>)
10.	<primary>	::=	id
11.	<primary>	::=	intliteral
12.	<addop> ::= +			
13.	<addop> ::= -			

1) A program is of the form

```
begin
  statements
end
```

2) Permissible statements:
• assignment read (list of variables)
• write (list of expressions)

3) Variables are declared implicitly their type is integer

4)Statements end in a semi-colon

5) Valid operators are +, - but can use parentheses

BNF

- Items such as <`program`> are non-terminals
  - require further expansion
- Items such as `begin` are terminals
  - correspond to language tokens
- Usual to combine productions using | (or)
  - e.g. <primary> ::= (<expr>) | id | intliteral

Parsing

- Bottom-up
  - Look for patterns in the input which correspond to phrases in the grammar
  - Replace patterns of items by phrases, then combine these into higher-level phrases, and so on
  - Stop when input converted to single <program>
- Top-down
  - Assume input is a <program>
  - Search for each of the sub-phrases forming a <program>, then for each of the sub-sub-phrases, and so on
  - Stop when we reach terminals
- A program is syntactically correct iff it can be derived from the CFG

Question

- Consider the following grammar, where S, A and B are non-terminals, and a and b are terminals:

  S ::= AB

  A ::= a

  A ::= BaB

  B ::= bbA

- Which of the following is FALSE?

  a) The length of every string derived from S is even.
  b) No string derived from S has an odd number of consecutive b's.
  c) No string derived from S has three consecutive a's.
  d) No string derived from S has four consecutive b's.
  e) Every string derived from S has at least as many b's as a's.

Answer:d
No string derived from S has four consecutive b's

Example

Parse: `begin A := B + (10 - C); end`

```
<program>
begin <stat-list> end                          (apply rule 1)
begin <statement> end                                     (2)
begin id := <expr> ; end                                  (3)
begin id := <primary> <addop> <primary>; end    (8)
begin id := <primary> + <primary> ; end                  (12)
...
```

Exercise

- Complete the previous parse

- Clue - this is the final line of the parse:

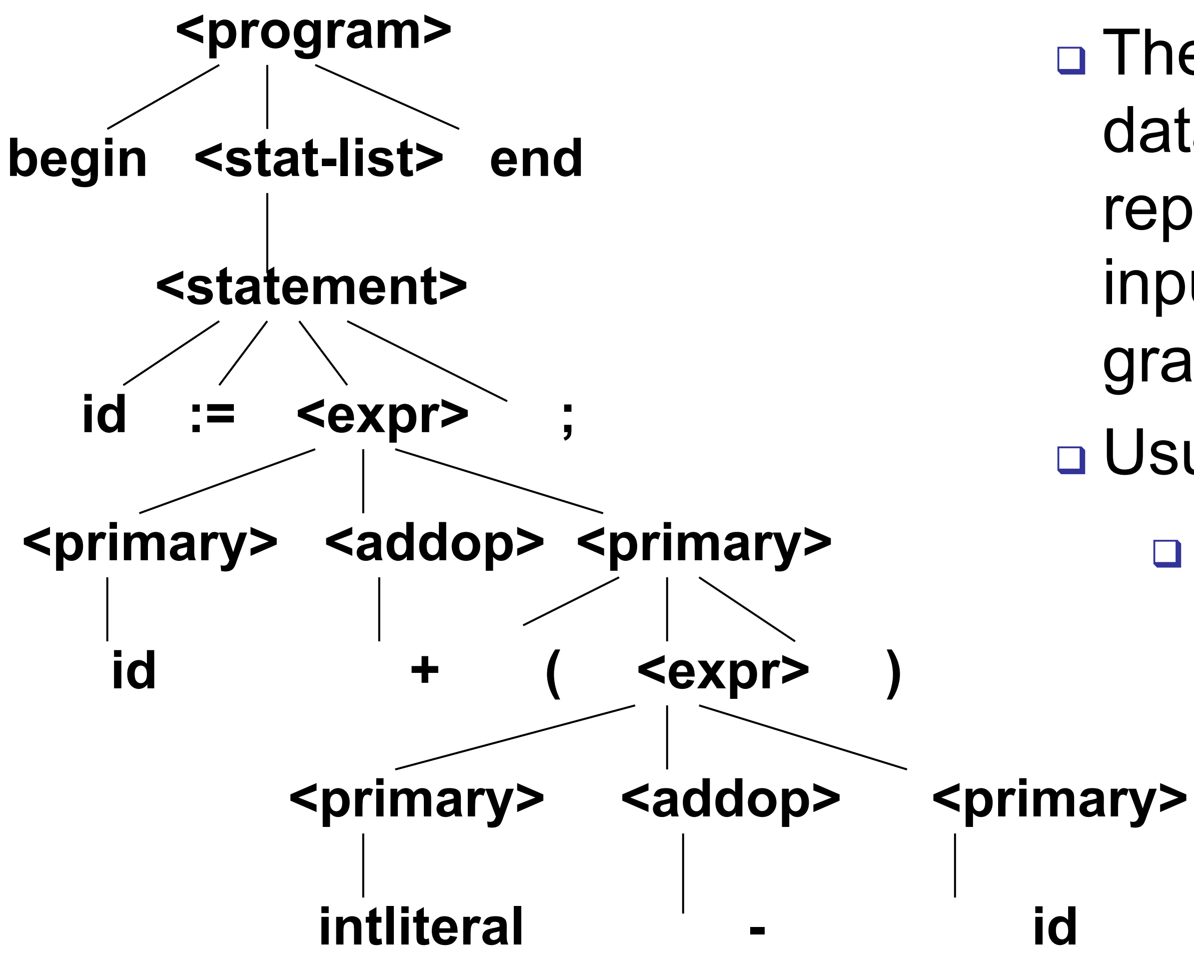
```
begin id := id + (intliteral - id); end
```

Answer

- Parse `begin A := B + (10 - C); end`

<program>	
begin <stat-list> end	(apply rule 1)
begin <statement> end	(2)
begin id := <expr> ; end	(3)
begin id := <primary> <addop> <primary>; end	(8)
begin id := <primary> + <primary> ; end	(12)
begin id := id + <primary> ; end	(10)
begin id := id + (<expr>) ; end	(9)
begin id := id + (<primary><addop><primary>); end	(8)
begin id := id + (<primary> - <primary>); end	(13)
begin id := id + (intliteral - <primary>); end	(11)
begin id := id + (intliteral - id); end	(10)

Parse Tree

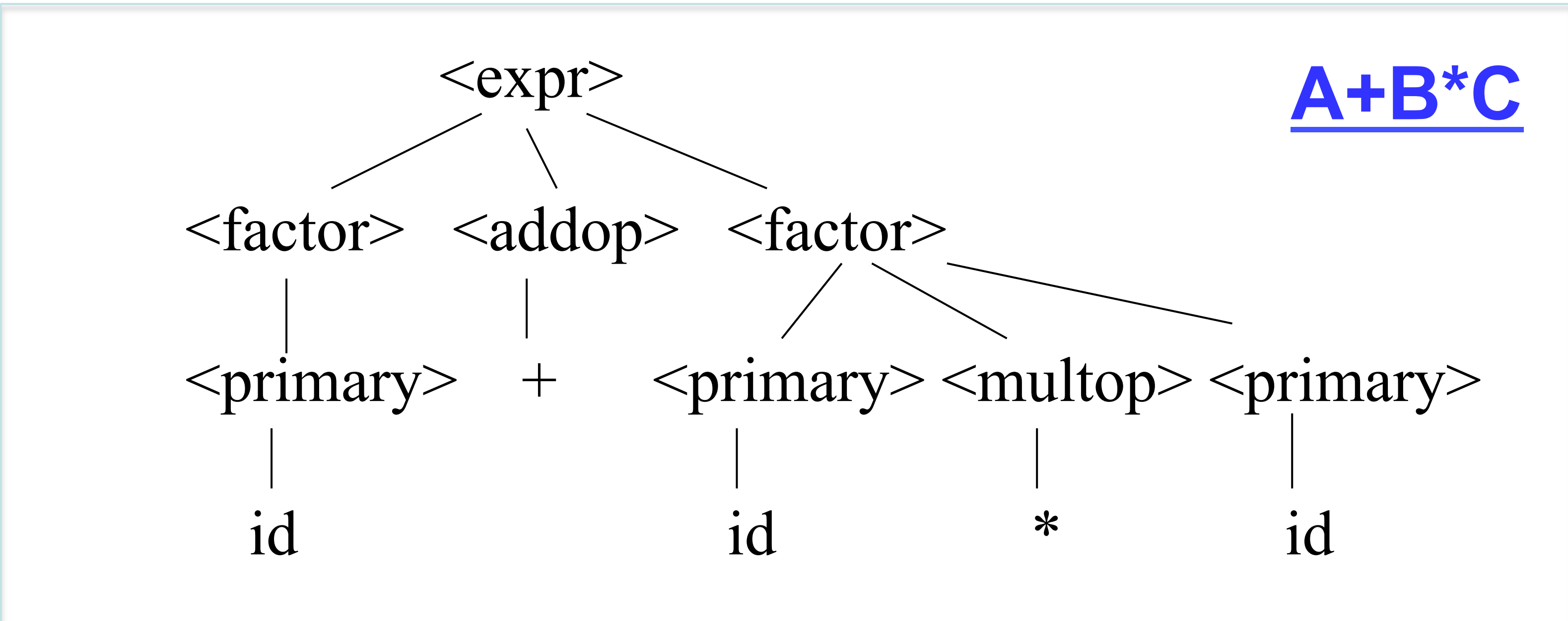


- The parser creates a data structure representing how the input is matched to grammar rules.
- Usually as a *tree.*
  - *Also called syntax tree or derivation tree*

Expression Grammars

- For expressions, a CFG can indicate associativity and operator precedence, e.g.

```
  <expr> ::= <factor> { <addop> <factor> }
 <factor> ::= <primary> { <multop> <primary> }
<primary> ::= ( <expr> ) | id | literal
```



Ambiguity

- A grammar is *ambiguous* if there is more than one parse tree for a valid sentence.

- Example:

```
expr =>  expr + expr
       | expr * expr
       | id
    | number
```

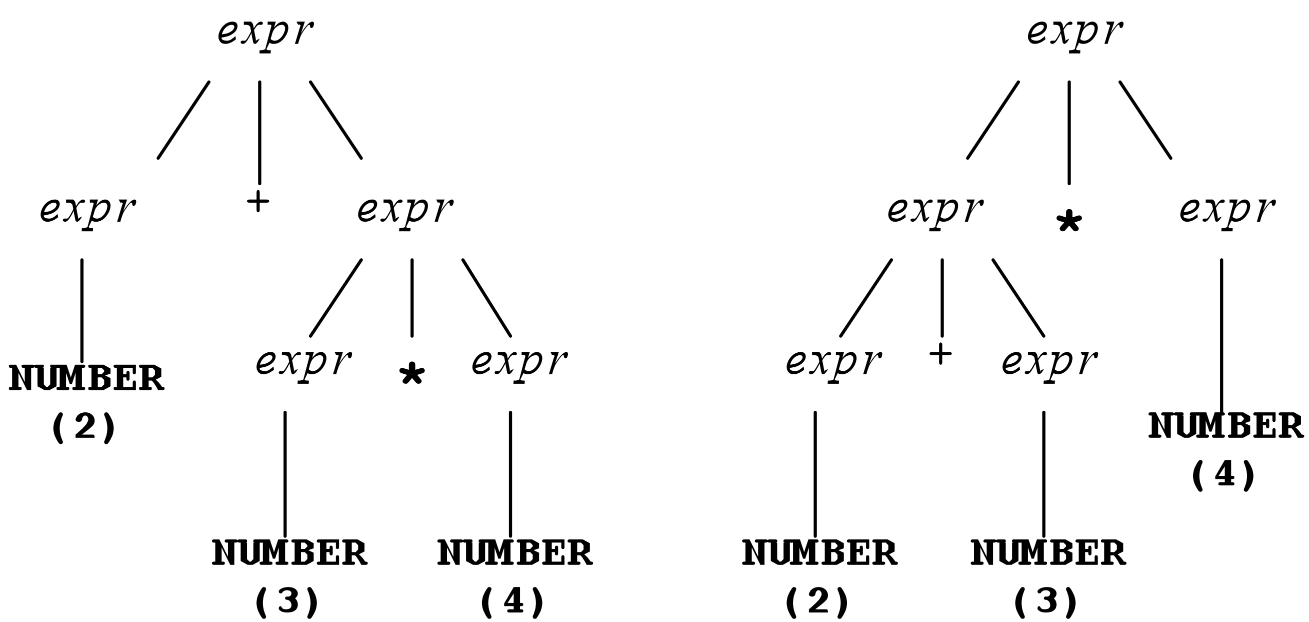
- How would you parse **x + y * z** using this rule?

Example of Ambiguity

- Grammar Rules:

  expr **=>** ***expr*** + ***expr*** **|** ***expr*** * ***expr***
  | (***expr***) **|** NUMBER

- Expression: 2 + 3 * 4
- Two possible parse trees:

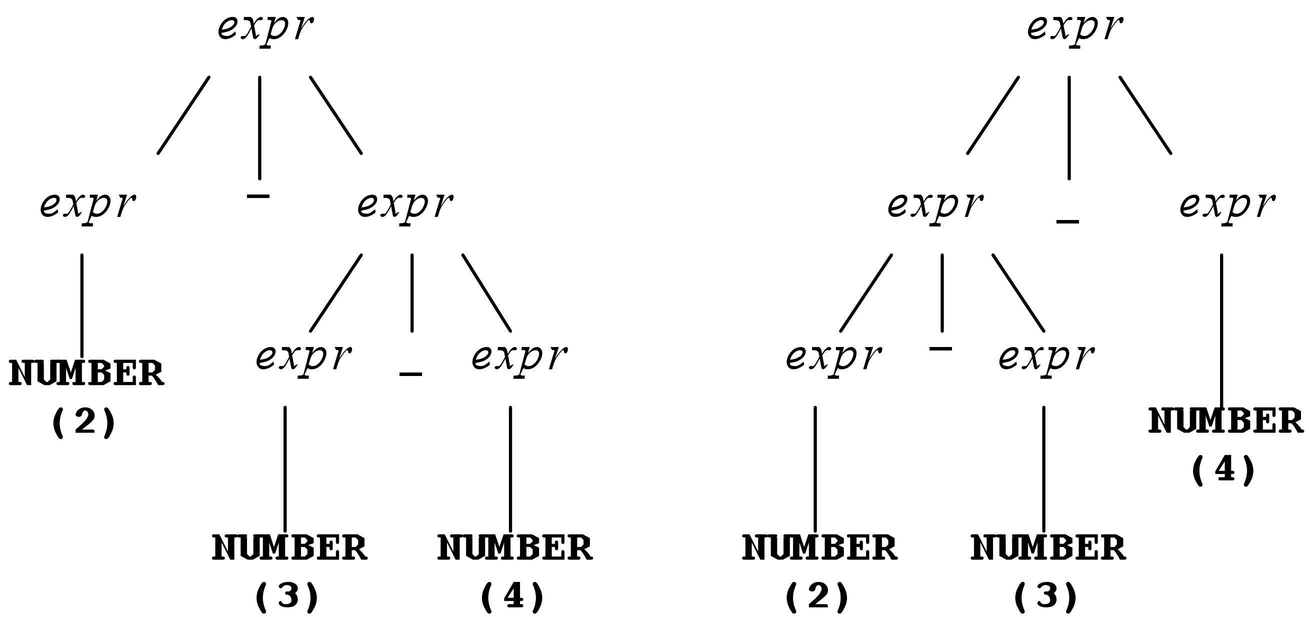


Another Example of Ambiguity

- Grammar rules:

expr **=>** ***expr*** + ***expr*** **|** ***expr*** − ***expr***
| (***expr***) **|** NUMBER

- Expression: 2 − 3 − 4
- Parse trees:



Ambiguity

- Ambiguity can lead to inconsistent implementations of a language.
  - Ambiguity can cause infinite loops in some parsers.
  - Specification of a grammar should be <u>un</u>ambiguous!
- How to resolve ambiguity:
  - rewrite grammar rules to remove ambiguity
  - add some additional requirement for parser, such as "always use the left-most match first"
  - EBNF (later) helps remove ambiguity

Abstract Syntax Tree (AST)

- More compact form of derivation tree
  - contains just enough info. to drive later phases
    e.g. Y := 3*X + I