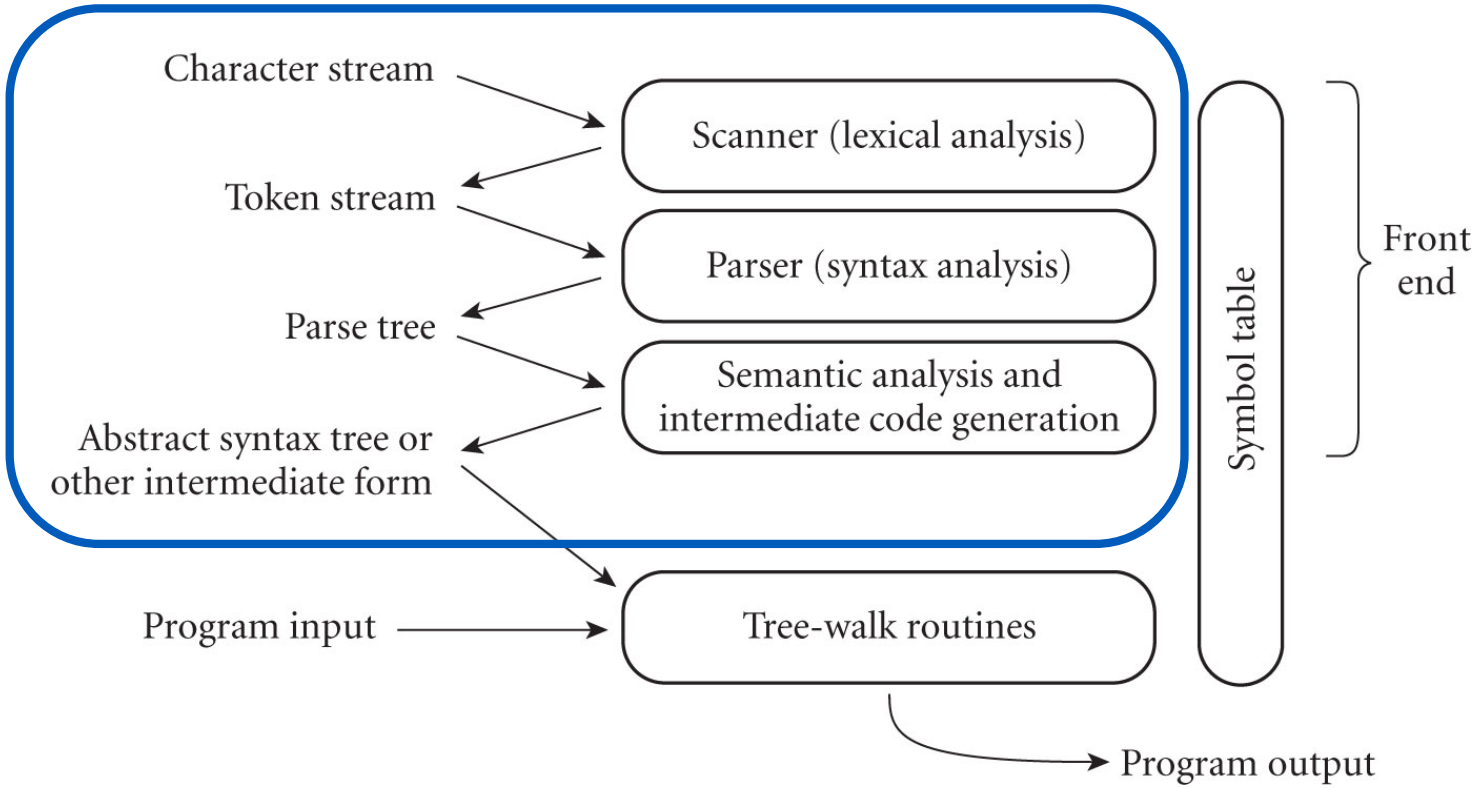
CS 320 : Continuing on Formal Grammars

Marco Gaboardi
CDS 1019
gaboardi@bu.edu

Parsing and semantic analysis



Generator vs Recognizer

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

Recognize a sentence

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

Generate a sentence

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

Some of the challenges:

- There is a (potentially) infinite number of source programs that we need to recognize.
  - An infinity of words
  - An infinity of sentences
- There should be no ambiguity in the way the program is interpreted.
  - Unique vocabulary,
  - Uniquely determine sentences
- The source program may contain syntax errors and the compiler/interpreter has to recognize them.
  - Lexical errors (errors in the choice of words)
  - Grammatical errors (errors in the construction of sentences)

Is a BNF grammar specific enough for an interpreter to execute it?

Here a simple grammar for expressions:

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

How shall the interpreter/compiler execute the following expression?

```
2+3*4
```

This can be interpreted as

```
(2+3)*4
```

or as

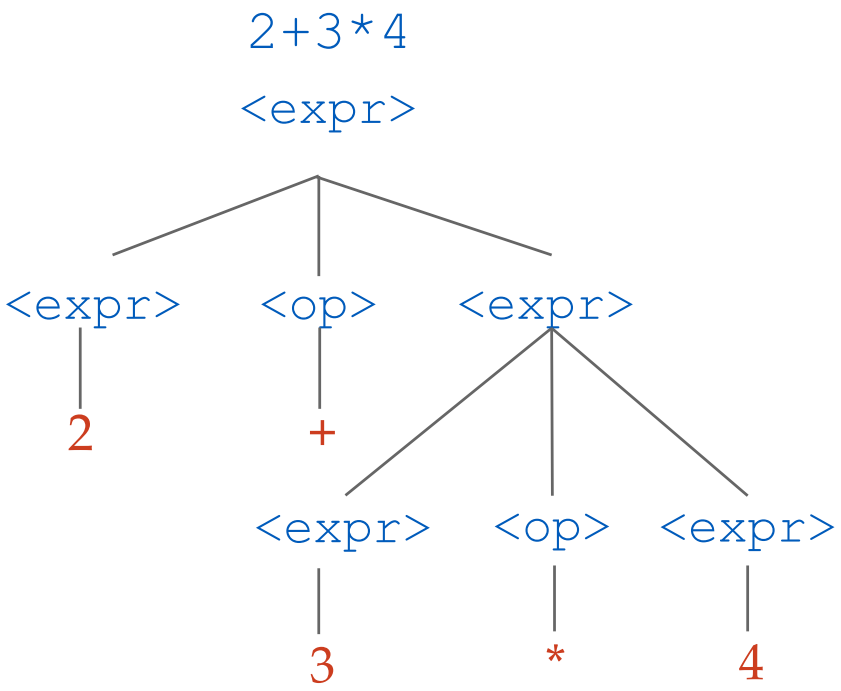
```
2+(3*4)
```

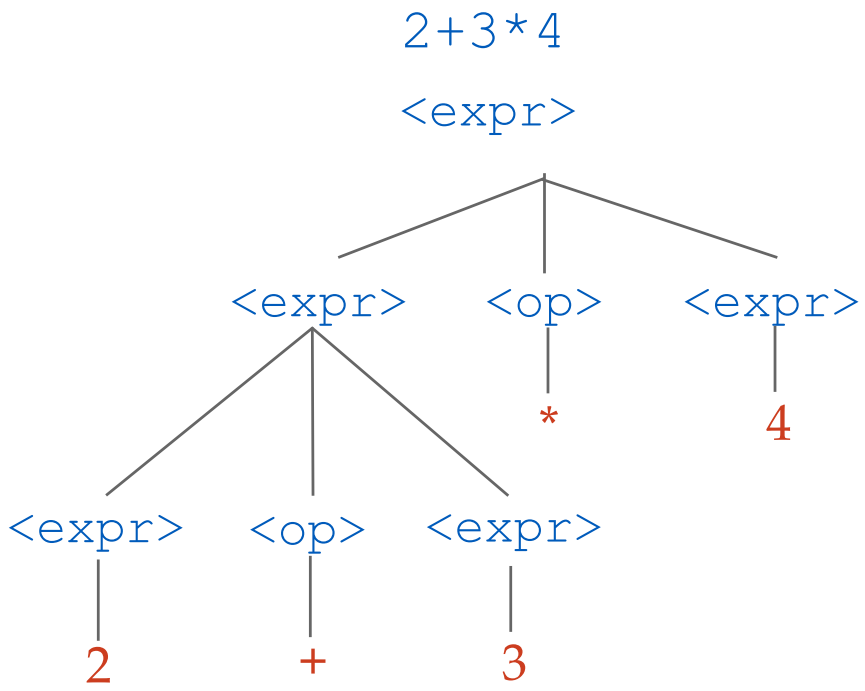
Note: here the parenthesis are just to show the possible ambiguity, they are not part of the grammar.

Ambiguous Grammars

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

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```





Ambiguous Grammars

Ambiguous grammars are, in general, undesirable in formal languages.

Why?

It makes parsing difficult – and more error prone.

Ambiguity can have different sources.

Good news: we can usually eliminate the ambiguity by revising the grammar.

Some examples

```
<funtype> ::= <type> | <funtype> -> <funtype>
<type> ::= int | float | bool
```

Is this grammar ambigous? Show why?

Some examples

<prodtype> ::= int * <prodtype> | int

Is this grammar ambigous? Show why?

Some examples

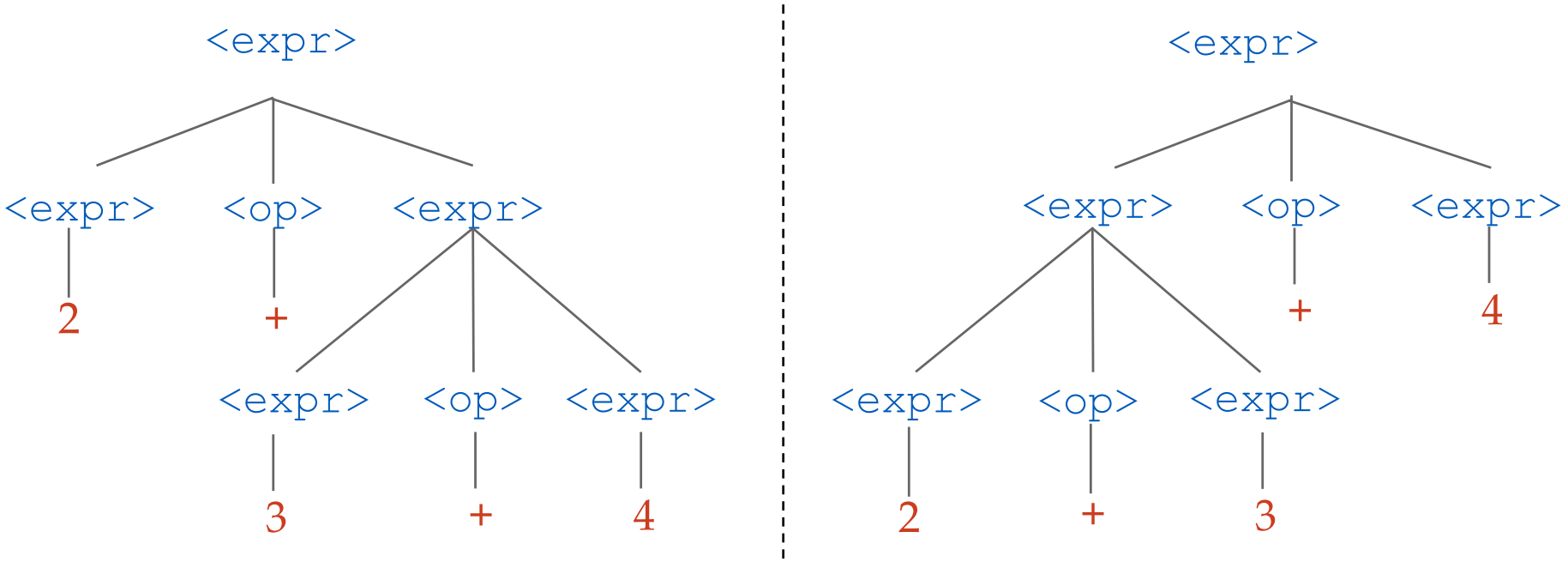
```
<expr> ::= <expr> + <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
```

Is this grammar ambigous? Show why?

How can we avoid ambiguity?

How can we disambiguate between the two parse trees for the following expression?

2+3+4



How can we avoid ambiguity?

How can we disambiguate between the two parse trees for the following expression?

2+3+4

First idea: make the parentheses part of the language

((2+3)+4)

(2+(3+4))

We need to add them everywhere!

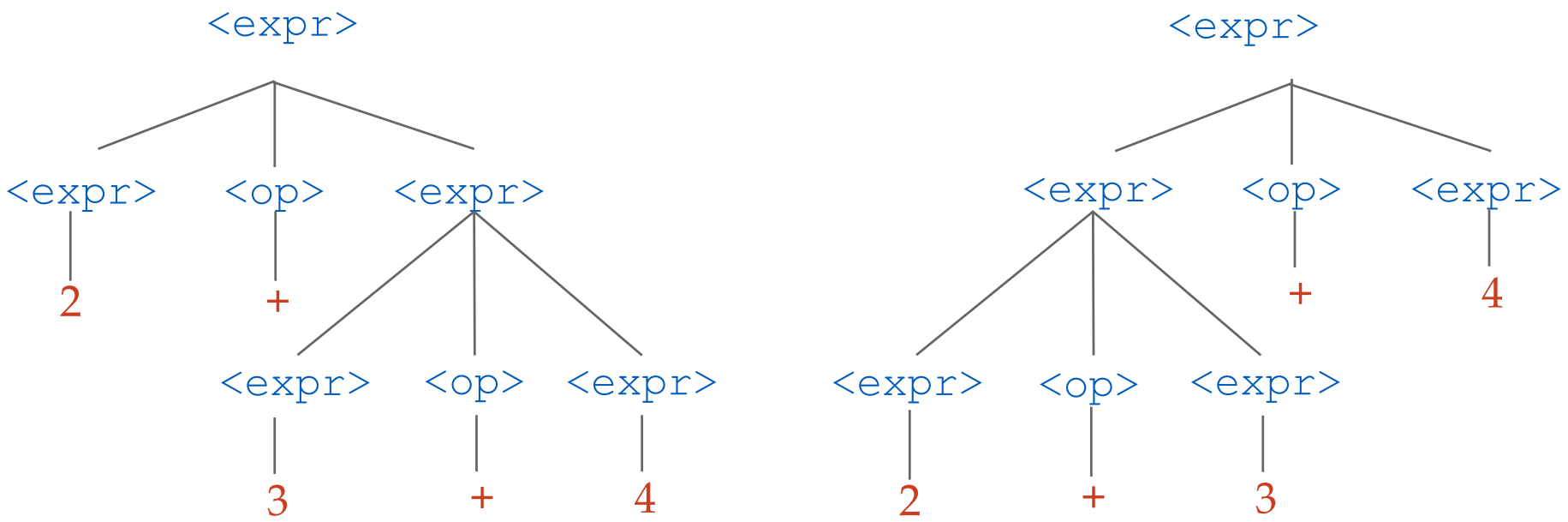
One way to do this is to change the grammar:

```
<expr> ::= ( <expr> <op> <expr> )
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

We add parentheses around every expression

How can we avoid ambiguity and preserve the structure of the grammar?

Second idea: If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.



Problem: it requires to work directly with parse trees.

How can we avoid ambiguity and preserve the structure of the grammar?

Why is the previous grammar ambiguous?

```
2+3*4
```

Two “classes” of operations that have different precedence and the grammar does not distinguish them.

```
2+3+4
```

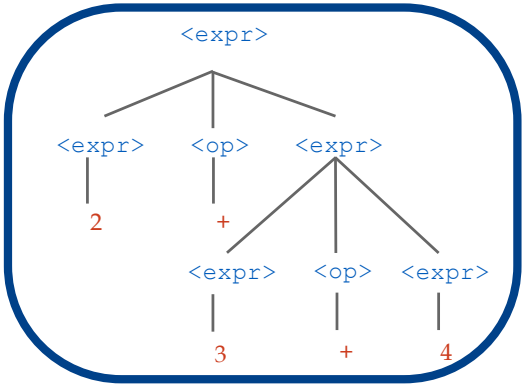
Two “occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

Dealing with associativity?

2+3+4

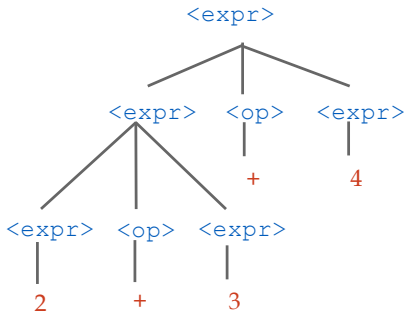
Two “occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```



We need to break the symmetry and commit to one choice.

```
<expr> ::= <const>|<const><op><expr>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```



Dealing with associativity?

```
<expr> ::= <const>|<const><op><expr>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>    ::= +|-|*|/
```

We use two nonterminal to break the symmetry

How can we derive the following expression?

2+3+4+5

```
<expr> => <const> <op> <expr>
       => 2 <op> <expr>
        => 2 + <expr>
       => 2 + <const> <op> <expr>
        => 2 + 3 <op> <expr>
        => 2 + 3 + <expr>
        => 2 + 3 + <const> <op> <expr>
        => 2 + 3 + 4 <op> <expr>
        => 2 + 3 + 4 + <expr>
        => 2 + 3 + 4 + <const>
        => 2 + 3 + 4 + 5
```

Dealing with associativity?

```
<expr> ::= <const>|<const><op><expr>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

We use two nonterminal to break the symmetry

How can we recognize the following expression?

2+3+4+5

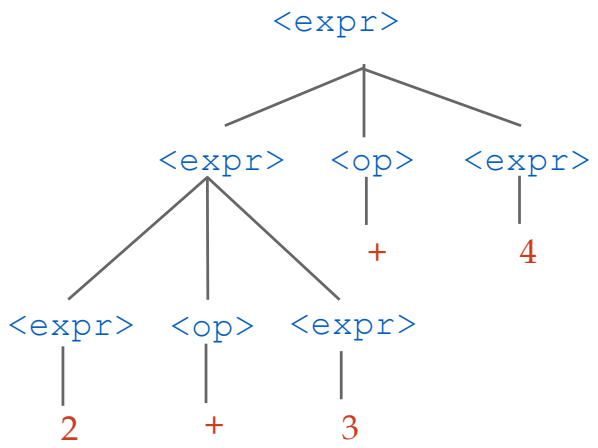
Dealing with associativity?

```
<expr> ::= <const>|<const><op><expr>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

We use two nonterminal to break the symmetry

How can we implement it?

Associativity by Grammar Design



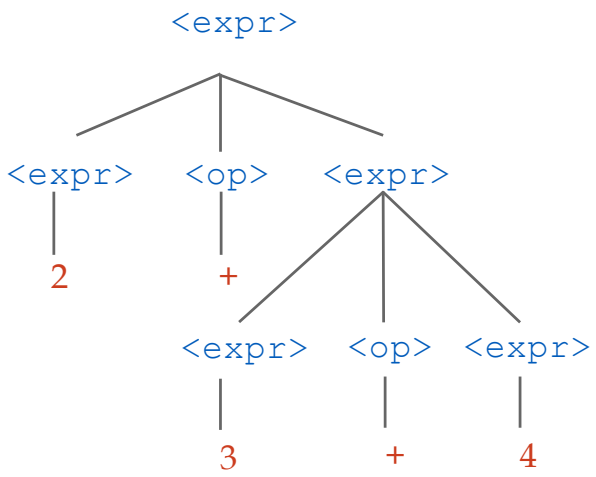
Left-associative

```
<expr> ::= <const>|<expr><op><const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +
```

Left-recursive

Right-associative

```
<expr> ::= <const|<const><op><expr>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +
```



Right-recursive