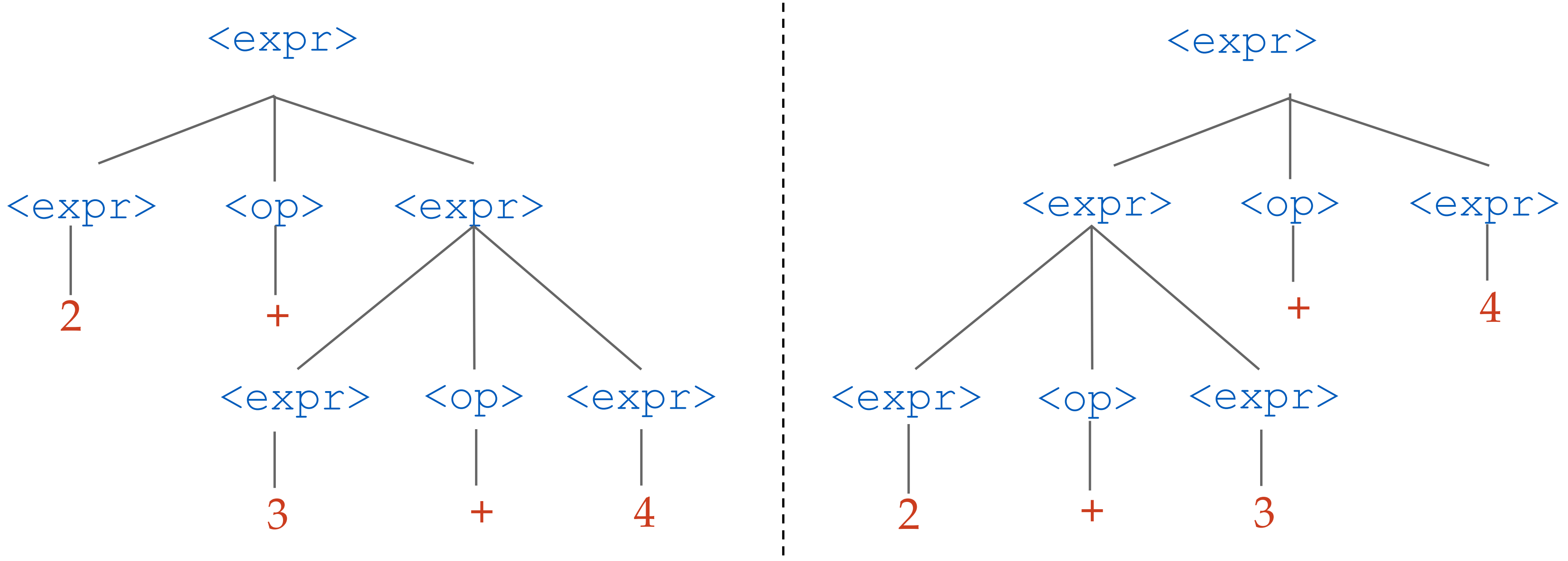
CS 320 : Continuing on Formal Grammars

Marco Gaboardi
CDS 1019
gaboardi@bu.edu

How can we avoid ambiguity?

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

2+3+4



Dealing with associativity?

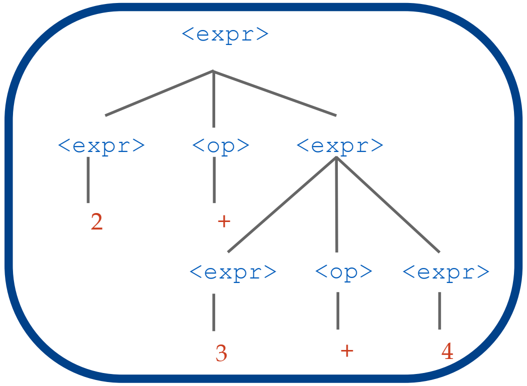
2+3+4

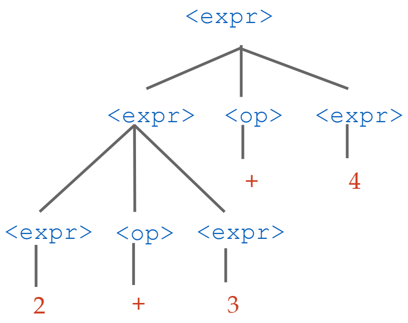
Two "occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

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

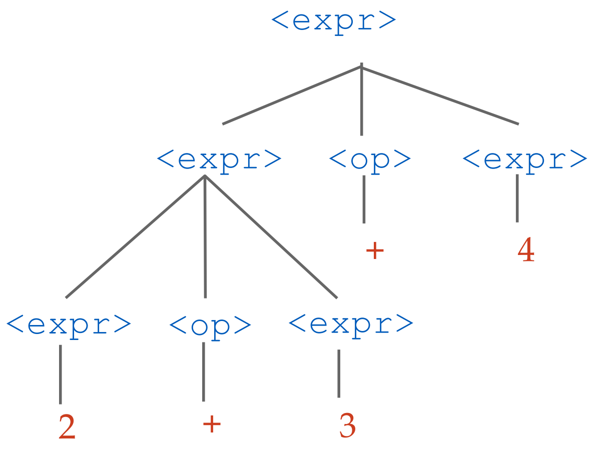
We need to break the symmetry and commit to one choice.

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





Associativity by Grammar Design



Left-associative

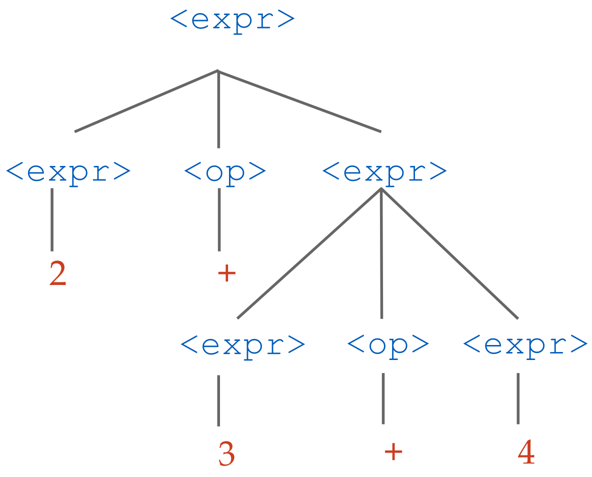
```
<expr>  ::= <const>|<expr><op><const>
<const>  ::= 1|2|3|4|5|6|7|8|9|0
<op>    ::= +
```

Left-recursive

Right-associative

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

Right-recursive



Dealing with associativity?

```
<expr>  ::= <const>|<expr><op><const>
<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?

Some examples

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

Design an equivalent grammar which is right associative.

Some examples

```
<expr> ::= <atomic_expr> | <expr> <expr>
<atomic_expr> ::= f | a | b
```

Design an equivalent grammar which is right associative.

Dealing with precedence?

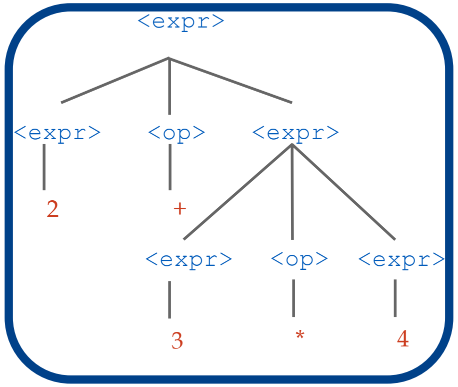
2+3*4

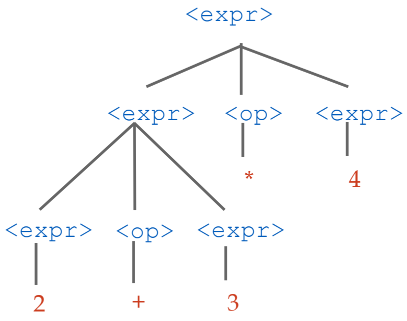
Two “classes” of operations that have different precedence and the grammar does not distinguish them.

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

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

```
<expr> ::= <expr> <addop> <term>
           | <term>
<term> ::= <term> <mulop> <term>
           | <const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>   ::= +
<mulop>   ::= *
```





Dealing with precedence?

```
<expr> ::= <expr> <addop> <term>
         | <term>
<term> ::= <term> <mulop> <term>
         | <const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>   ::= +
<mulop>   ::= *
```

We use two nonterminal to break the symmetry

How can we derive the following expression?

2+3*4

```
<expr> => <expr> <addop> <term>
       => <term> <addop> <term>
       => <const> <addop> <term>
       => 2 <addop> <term>
       => 2 + <term>
       => 2 + <term> <mulop> <term>
       => 2 + 3 <mulop> <term>
       => 2 + 3 * <term>
       => 2 + 3 * 4
```

Dealing with precedence?

```
<expr> ::= <expr> <addop> <term>
           | <term>
<term> ::= <term> <mulop> <term>
           | <const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>    ::= +
<mulop>    ::= *
```

We use two nonterminal to break the symmetry

Can we derive the following expression?

`(2+3)*4`

Recovering general expressions

```
<expr> ::= <expr> <addop> <term>
           | <term>
<term> ::= <term> <mulop> <term>
           | <const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>    ::= +
<mulop>    ::= *
```

Can we derive the following expression?

`(2+3)*4`

We need to introduce parentheses.

```
<expr> ::= <expr> <addop> <term>
           | <term>
<term> ::= <term> <mulop> <term>
           | <factor>
<factor> ::= <const> | ( <expr> )
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>    ::= +
<mulop>    ::= *
```

Dealing with precedence?

```
<expr> ::= <expr> <addop> <term>
         | <term>
<term> ::= <term> <mulop> <term>
         | <factor>
<factor> ::= <const> | ( <expr> )
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>   ::= +
<mulop>   ::= *
```

Can we derive the following expression?

(2+3)*4

```
<expr> => <term>
       => <term> <mulop> <term>
       => <factor> <mulop> <term>
       => ( <expr> ) <mulop> <term>
       => ( <expr> <addop> <term> ) <mulop> <term>
       => ( <factor> <addop> <term> ) <mulop> <term>
       => ( <const> <addop> <term> ) <mulop> <term>
       => ( 2 <addop> <term> ) <mulop> <term>
       => ( 2 + <term> ) <mulop> <term>
       => ( 2 + 3 ) <mulop> <term>
       => ( 2 + 3 ) * <term>
       => ( 2 + 3 ) * 4
```

Putting everything together

```
<expr> ::= <expr> <addop> <term>
         | <term>
<term> ::= <term> <mulop> <term>
         | <factor>
<factor> ::= <const> | ( <expr> )
<const> ::= 1|2|3|4|5|6|7|8|9|0
<addop>    ::= + | -
<mulop>    ::= * | /
```

Is this grammar still ambiguous?

No magic wand, we have to determine whether we can build two parse trees for the same expression. So, we need to look at the parse trees corresponding to its derivations.

Summary

Goals

- To understand how programs statements are recognized by the interpreter/compiler.
- To understand how the sentential structure of a program can be described as a data structure through the use of Formal Grammars.

In concrete:

How to design an unambiguous grammar with precedence and associativity.

Associativity by Grammar Design

Ambiguous

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

Unambiguous

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

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

Associativity by Grammar Design

Ambiguous

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

Unambiguous

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

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

Difficult to implement using parser combinators.