CS 320 : Formal Grammars

Marco Gaboardi
MSC 116
gaboardi@bu.edu

Announcements

- Third theory assignment posted soon.

From previous classes…

BNF - Backus Normal/Naur Form

- A grammar is defined by a set of terminals (tokens), a set of nonterminals, a designated **nonterminal start symbol**, and a finite nonempty set of rules

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

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
```

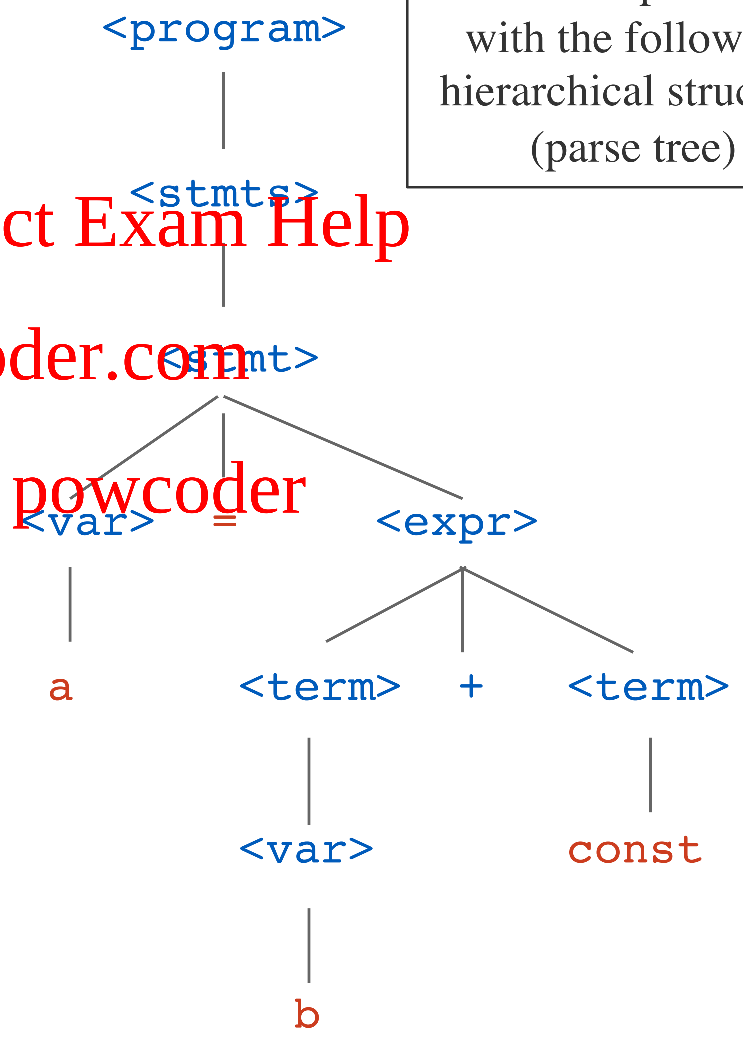
Some of the challenges:

- 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)

Parse Tree

- A parse tree is a hierarchical representation of a derivation

We can represent it with the following hierarchical structure (parse tree)



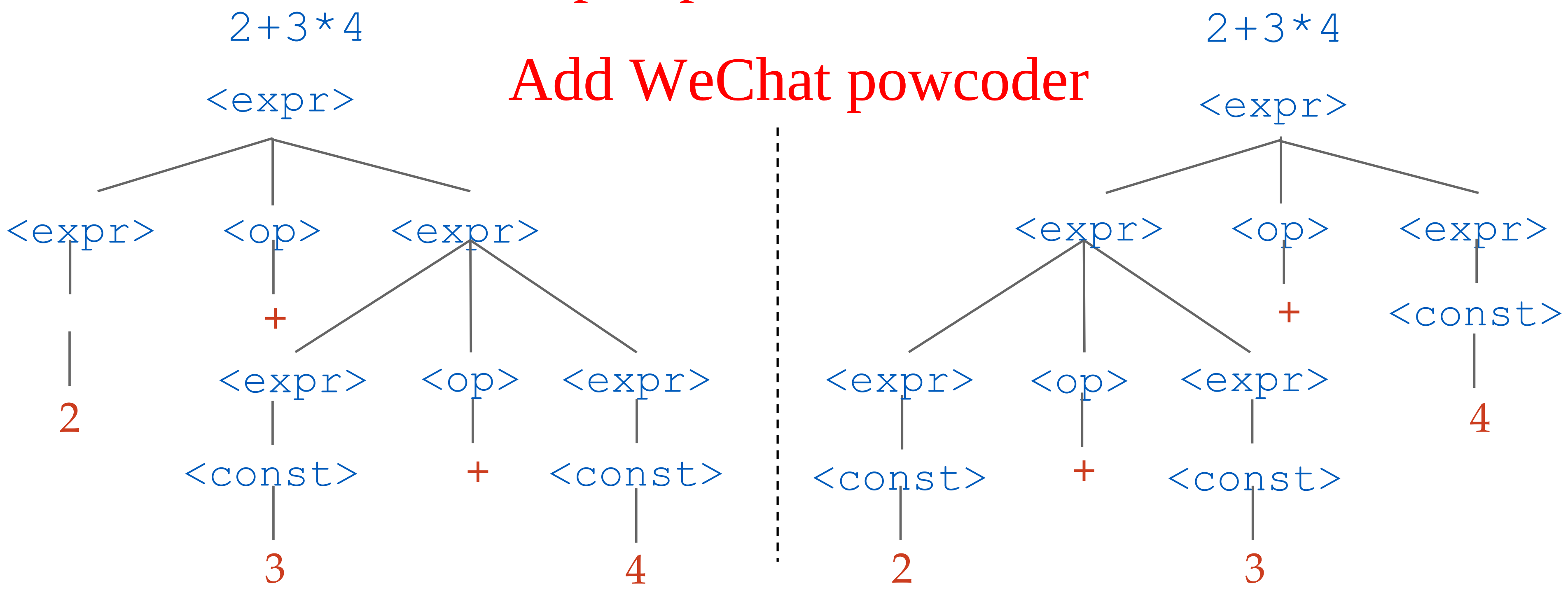
Suppose we have the following derivation

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

Ambiguous Grammars

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

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



Plan for today

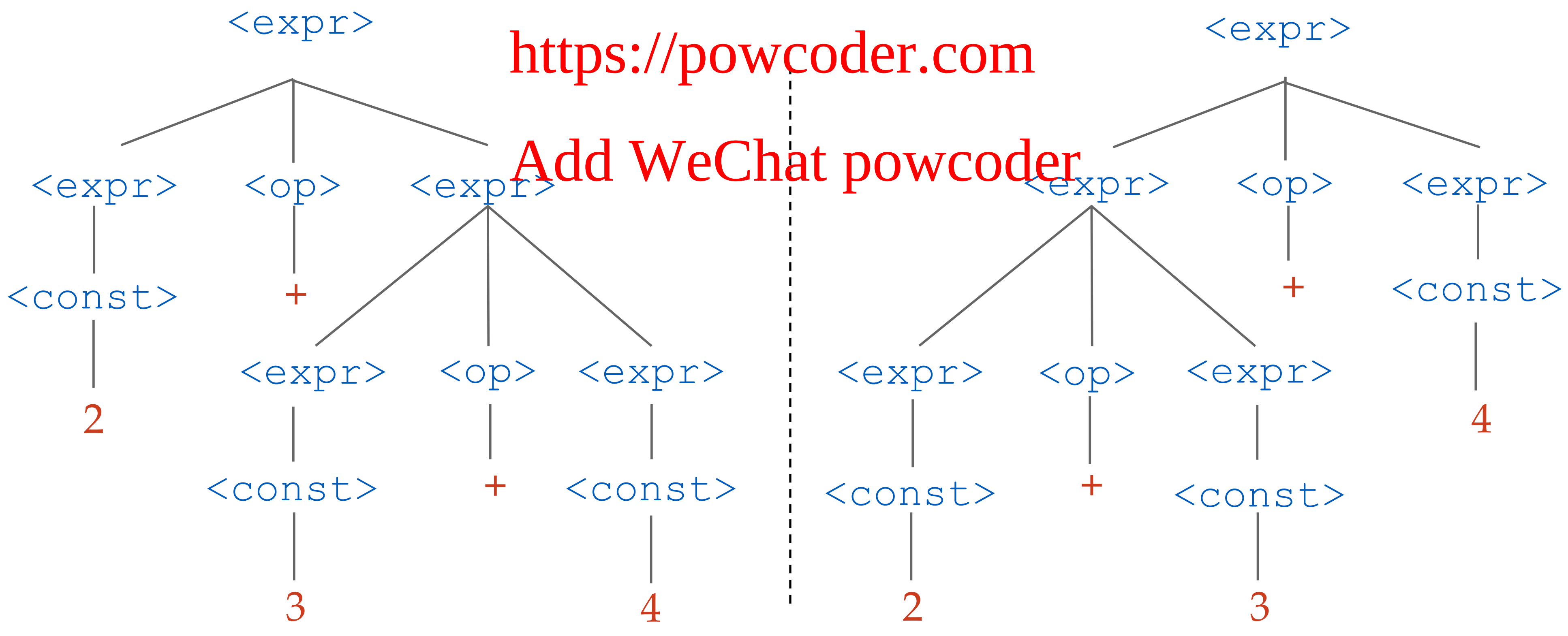
Disambiguate ambigous grammars

How can we avoid ambiguity?

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

2+3+4

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



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!

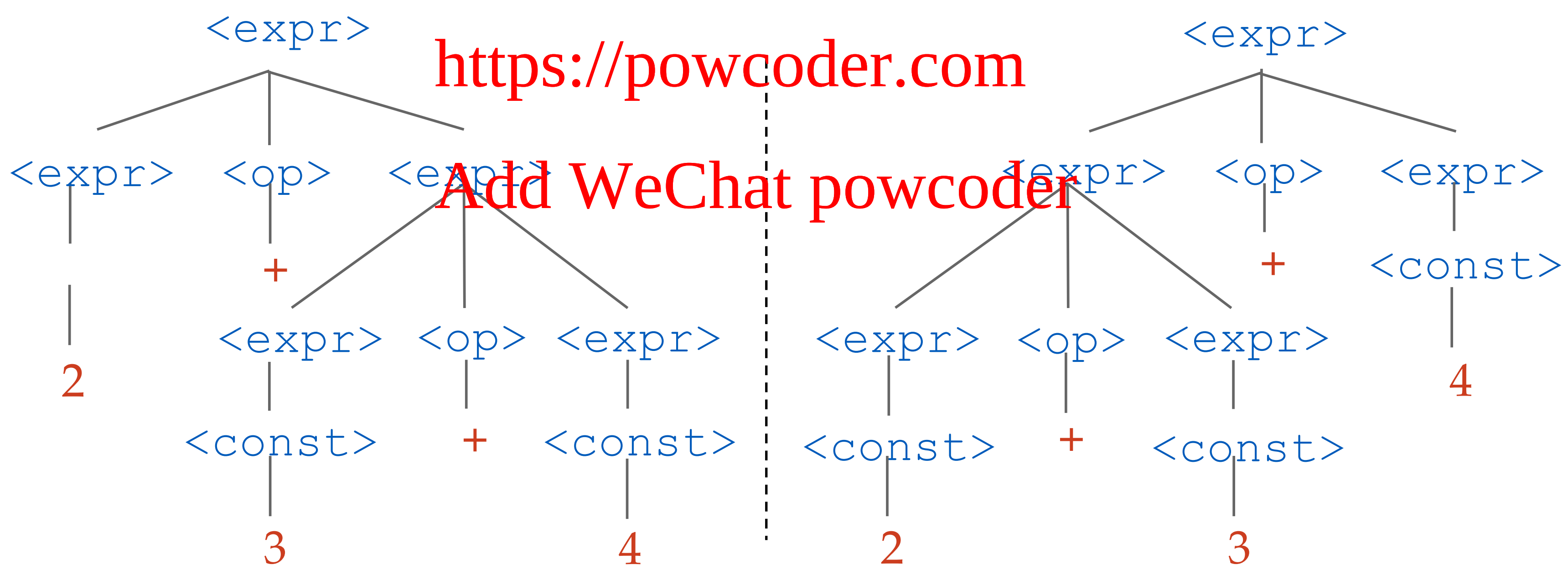
One way to do this is to change the grammar:

```
<expr>  ::= <const> | ( <expr> <op> <expr> )
<const> ::= 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

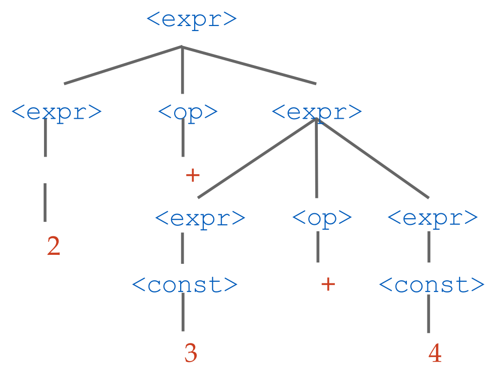
Two “occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

Dealing with associativity?

2+3+4

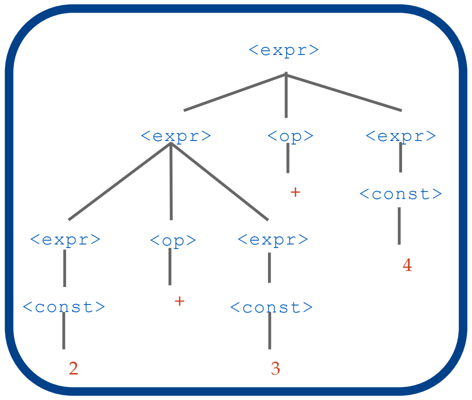
Two "occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

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



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

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



Dealing with associativity?

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

We modify recursion to break the symmetry

How can we derive the following expression?

2+3+4+5

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

Dealing with associativity?

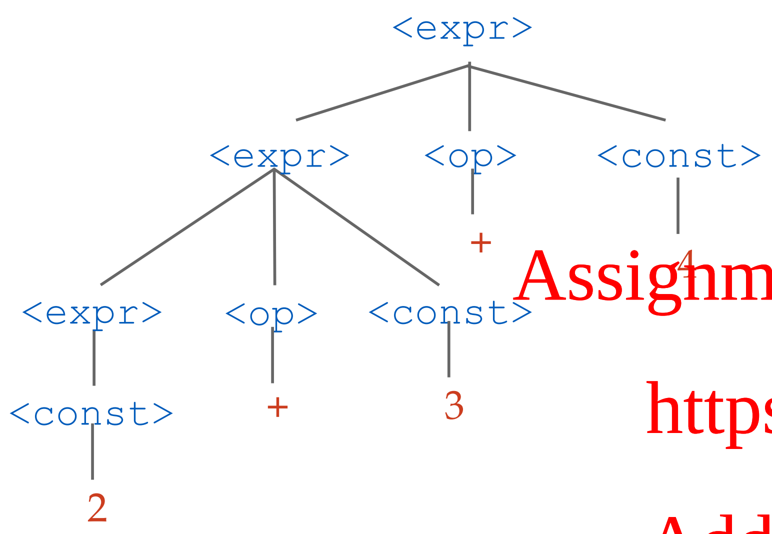
```
<expr>  ::= <const>|<expr><op><const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>    ::= +
```

We modify recursion to break the symmetry

How can we recognize the following expression?

2+3+4+5

Associativity by Grammar Design



Left-associative

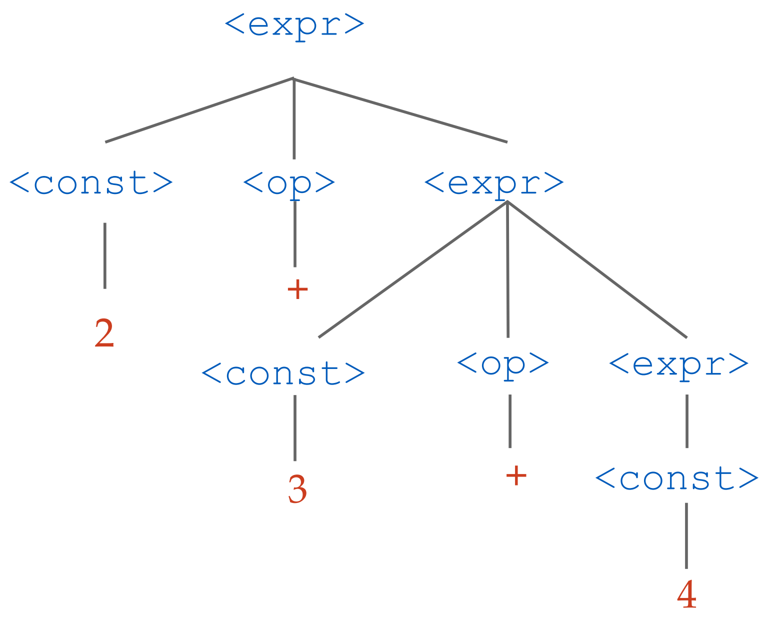
```
<expr>  ::= <const>|<expr><op><const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>    ::= +
```

Left-recursive

Right-associative

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

Right-recursive



Associativity by Grammar Design

Ambiguous

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

Unambiguous

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

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

TopHat Q1-Q5

Some examples

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

Design an equivalent grammar which is unambiguous and right associative.

Some examples

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

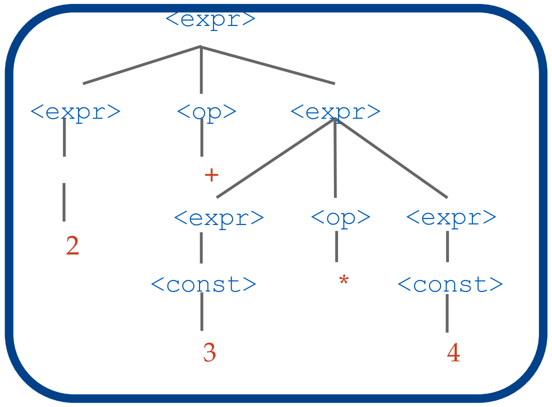
Design an equivalent grammar which is unambiguous and left associative.

Dealing with precedence?

2+3*4

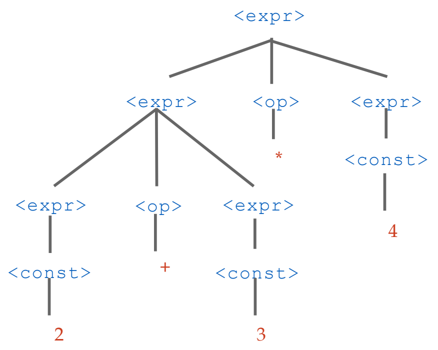
Two “classes” of operations that have different precedence and the grammar does not distinguish them.

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

How can we recognize the following expression?

2+3*4

Some examples

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

Design an equivalent grammar which is unambiguous and give precedence to * over ->.

Some examples

```
<expr> ::= <const> | <expr>#<expr>| <expr>$<expr>
<const> ::= f | a | b
```

Design an equivalent grammar which is unambiguous and give precedence to # over $.

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.

Some examples

```
<prog> ::= <expr> | <prog>;<prog> | <prog>@<prog>
<expr> ::= <var>++ | <var>-- | <expr>::<expr>
<var> ::= x | y | z
```

Design an equivalent unambigous grammar which is right associative on ::, which gives precedence to ; over @, and which allow to use begin … end to delimit programs with different precedence.

Some examples

```
<prog> ::= <expr> | <prog>;<prog> | <prog>@<prog>
<expr> ::= <var>++ | <var>-- | <expr>::<expr>
<var> ::= x | y | z
```

Design an equivalent unambigous grammar which is right associative on ::, which gives precedence to ; over @, and which allow to use begin … end to delimit programs with different precedence.

```
<prog> ::=  <seqprog> | <prog>@<seqprog>
<seqprog>::= <expr> | <prog>;<seqprog>
<expr> ::= <term>| <term>::<expr> | begin <prog> end
<term> ::= <var>++ | <var>--
<var> ::= x | y | z
```

TopHat Q8-Q13