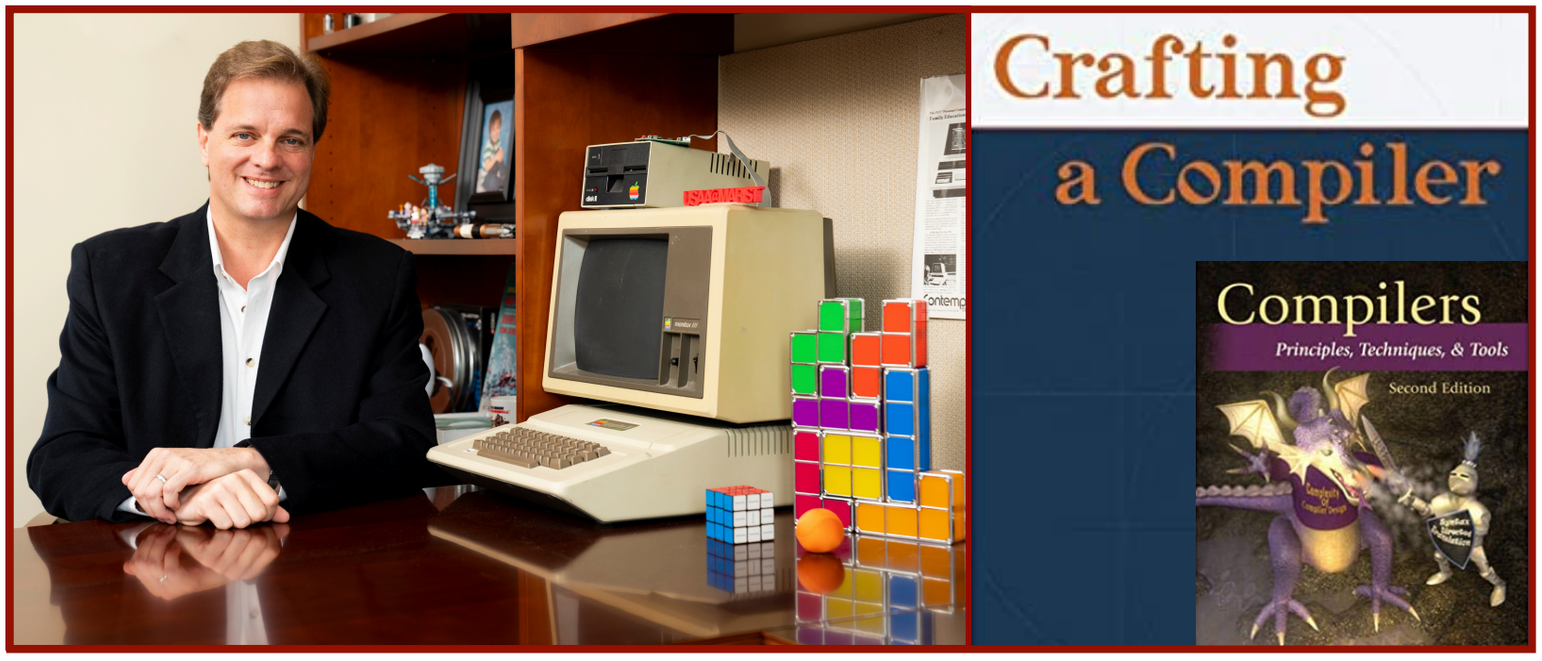
More Grammar Stuff



Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

LL(1) Analysis with First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1).

	nullable?	FIRST	FOLLOW													
<stmt>		{ }	{ }													
<expr>		{ }	{ }													
<term>		{ }	{ }													

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.          -> while <expr> do <stmt>
3.          -> <expr>;
4.  <expr> -> <term> := id
5.          -> zero? <term>
6.          -> not <expr>
7.          -> ++ id
8.          -> -- id
9.  <term> -> id
10.         -> const
```

Let's see whether or not this grammar is LL(1).

Pass #1a

Grab the terminals.

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while }	{ }												
<expr>		{ zero?, not, ++, -- }	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.          -> while <expr> do <stmt>
3.          -> <expr>;
4.  <expr> -> <term> := id
5.          -> zero? <term>
6.          -> not <expr>
7.          -> ++ id
8.          -> -- id
9.  <term> -> id
10.         -> const
```

Let's see whether or not this grammar is LL(1).

Pass #1b

FIRST(stmt) gets FIRST(expr).

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while, zero?, not, ++, -- }	{ }												
<expr>		{ zero?, not, ++, -- }	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1).

Pass #1c

FIRST(expr) gets FIRST(term).

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while, zero?, not, ++, -- }	{ }												
<expr>		{ zero?, not, ++, --, id, const}	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1).

Pass #2a

FIRST(stmt) gets FIRST(expr), which has changed.

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while, zero?, not, ++, -- id, const }	{ }												
<expr>		{ zero?, not, ++, --, id, const}	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1).

Pass #2b

FIRST(expr) gets FIRST(term), which has not changed.

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while, zero?, not, ++, -- id, const }	{ }												
<expr>		{ zero?, not, ++, --, id, const}	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1).

Pass #3 - no changes.

We're done.

What about nullable productions?

	nullable?	FIRST	FOLLOW												
<stmt>		{ if, while, zero?, not, ++, -- id, const }	{ }												
<expr>		{ zero?, not, ++, --, id, const}	{ }												
<term>		{ id, const }	{ }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

We don't need to look at the follow sets to develop the Parse Table because there are no null (epsilon) productions.

The contents of the follow sets are left as an exercise for the reader. You're welcome.

	nullable?	FIRST	FOLLOW												
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }												
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }												
<term>	*no*	{ id, const }	{ ? }												

First and Follow Sets

```
1.  <stmt> -> if <expr> then <stmt>
2.          -> while <expr> do <stmt>
3.          -> <expr>;
4.  <expr> -> <term> := id
5.          -> zero? <term>
6.          -> not <expr>
7.          -> ++ id
8.          -> -- id
9.  <term> -> id
10.         -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW		if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }													
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }													
<term>	*no*	{ id, const }	{ ? }													

LL(1) Analysis with a Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ **if**, while, zero?, not, ++, -- id, const }	{ ? }	1											
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }												
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, **while**, zero?, not, ++, -- id, const }	{ ? }	1		2									
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }												
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, **zero?, not, ++, -- id, const** }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }												
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, --, **id**, **const**}	{ ? }										4	4	
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ **zero?**, not, ++, --, id, const}	{ ? }					5					4	4	
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, **not**, ++, --, id, const}	{ ? }					5	6				4	4	
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, **++**, --, id, const}	{ ? }					5	6	7			4	4	
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, **--**, id, const}	{ ? }					5	6	7	8		4	4	
<term>	*no*	{ id, const }	{ ? }												

Parse Table

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }					5	6	7	8		4	4	
<term>	*no*	{ **id**, const }	{ ? }										9		

Parse Table

```
1.   <stmt> -> if <expr> then <stmt>
2.           -> while <expr> do <stmt>
3.           -> <expr>;
4.   <expr> -> <term> := id
5.           -> zero? <term>
6.           -> not <expr>
7.           -> ++ id
8.           -> -- id
9.   <term> -> id
10.          -> const
```

Let's see whether or not this grammar is LL(1) by making the Parse Table . . .

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }					5	6	7	8		4	4	
<term>	*no*	{ id, **const** }	{ ? }										9	10	

Parse Table - Proves LL(1)*ness*

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

The parse table contains no or 1 entry at every intersection of production and terminal. In other words, there are no cells in the parse table with multiple entries. So this grammar is explicit*, not vague, and therefore LL(1).

	nullable?	FIRST	FOLLOW	if	then	while	do	zero?	not	++	--	:=	id	const	;
<stmt>	*no*	{ if, while, zero?, not, ++, -- id, const }	{ ? }	1		2		3	3	3	3		3	3	
<expr>	*no*	{ zero?, not, ++, --, id, const}	{ ? }					5	6	7	8		4	4	
<term>	*no*	{ id, const }	{ ? }										9	10	

* as in “unambiguous”, if we were to overload that term.

Uncooperative Grammars

Not all grammars are as cooperative as this one.

- Some are ambiguous.
  - An ambiguous Context-Free Grammar is one where different derivations of the same code can produce different Concrete Syntax Trees.
  - This is bad and we'd like to avoid it.
  - We'll need to change the grammar to fix it.

```
1.  <stmt> -> if <expr> then <stmt>
2.         -> while <expr> do <stmt>
3.         -> <expr>;
4.  <expr> -> <term> := id
5.         -> zero? <term>
6.         -> not <expr>
7.         -> ++ id
8.         -> -- id
9.  <term> -> id
10.        -> const
```

- Some are not LL(1).
  - If it's not LL(1) then we cannot write a Recursive Descent Parser.
  - This is bad and we'd like to avoid it.
  - We'll need to change the grammar to fix it (assuming we want to write a Recursive Descent Parser). But there are other parsing possibilities…

Ambiguous Context-Free Grammars

Remember this **un**ambiguous grammar?

```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

input tokens: `9 - 5 + 2`

Even Earlier slide

```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

input tokens: 9 - 5 + 2

```
S. <list>
1. <list> + <digit>
2. <list> - <digit> + <digit>
3. <digit>- <digit> + <digit>
4.   9   -  <digit> + <digit>
4.   9   -    5    + <digit>
4.   9   -    5    +   2
```

Now what?
We're good! We've run out of non-terminals to turn into terminals at the same time as we've run out of input tokens to process. This is a beautiful thing... a successful parse.

Earlier slide

```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

input tokens: 9 - 5 + 2

```
S. <list>
   <list>            + <digit>
   list>           +   2
   list>  - <digit>+   2
   <list>   -   5  +   2
3. <digit>  -   5  +   2
4.    9     -   5  +   2
```


We've run out of non-terminals to turn into terminals at the same time as we've run out of input tokens to process. This is a beautiful thing... another successful parse, this time a right-most derivation.

What if we had a grammar a little less cooperative?

Ambiguous Context-Free Grammars

A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

Ambiguous Context-Free Grammars

A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

<expr>

Ambiguous Context-Free Grammars

A less cooperative grammar

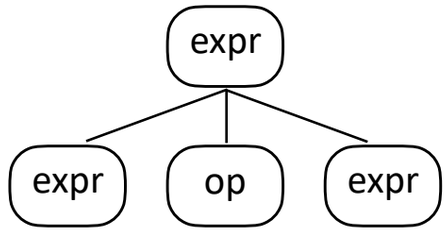
```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

<expr>

<expr> <op> <expr>



Ambiguous Context-Free Grammars

A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

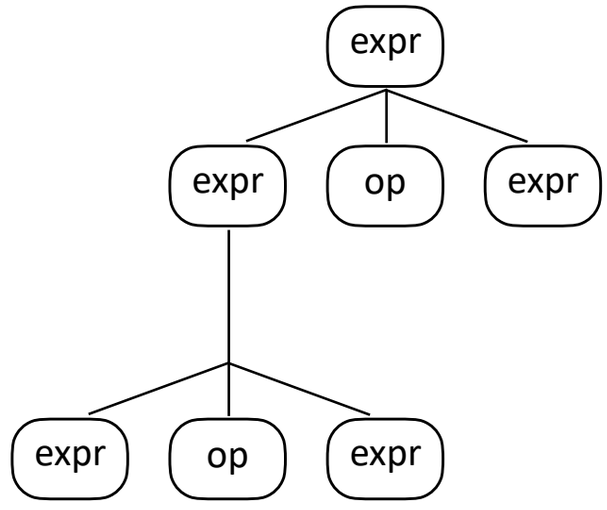
input tokens: `8 - 4 ÷ 2`

Left-most derivation

<expr>

<expr> <op> <expr>

<expr> <op> <expr> <op> <expr>



Ambiguous Context-Free Grammars

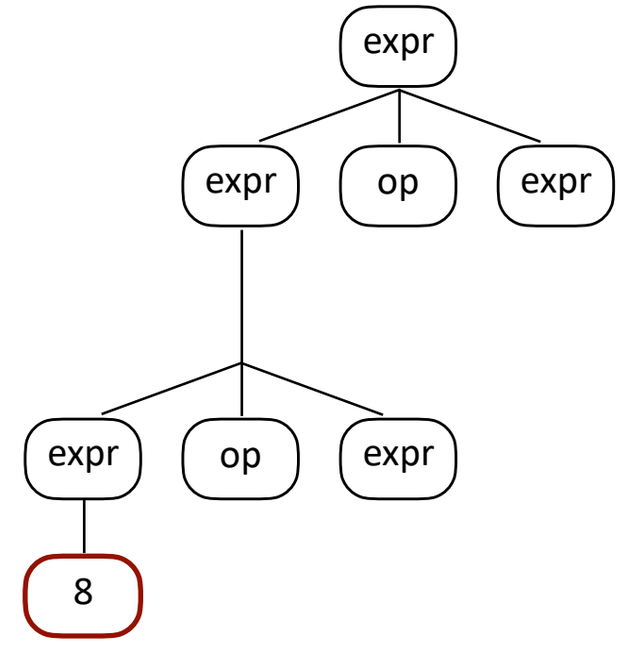
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
```



Ambiguous Context-Free Grammars

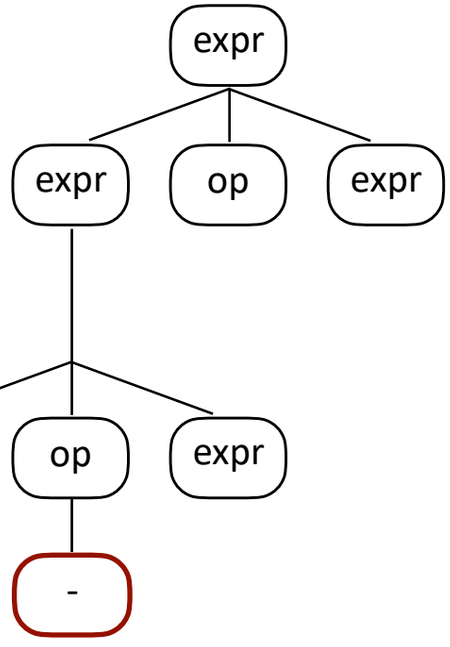
A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
```



Ambiguous Context-Free Grammars

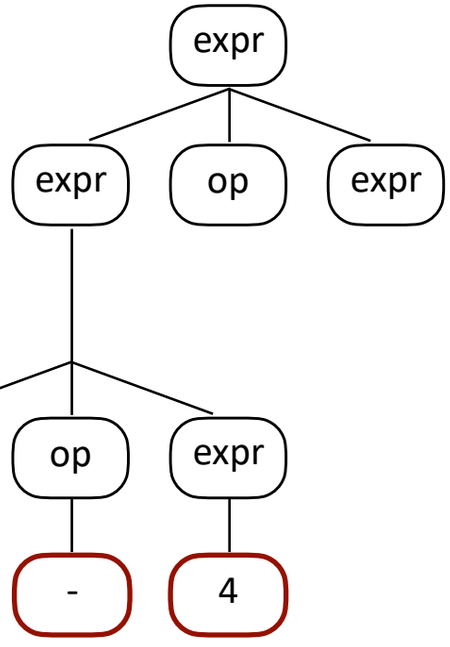
A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
```



Ambiguous Context-Free Grammars

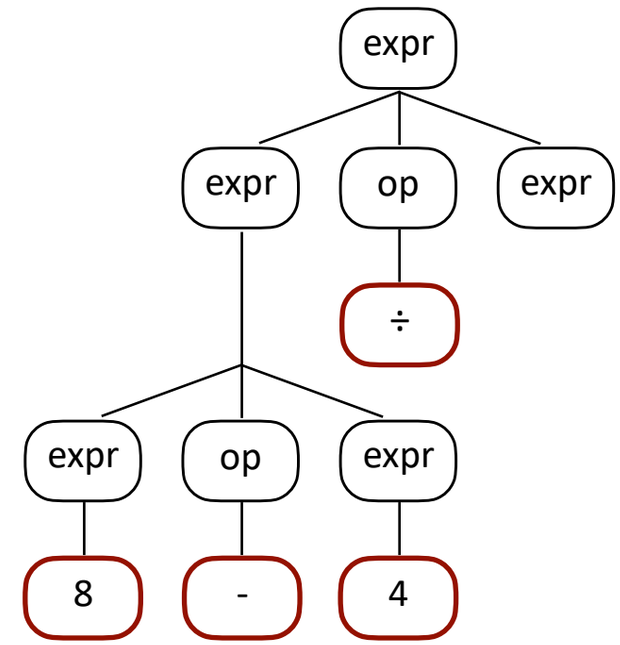
A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
[num,8]  -  [num,4]  ÷  <expr>
```



Ambiguous Context-Free Grammars

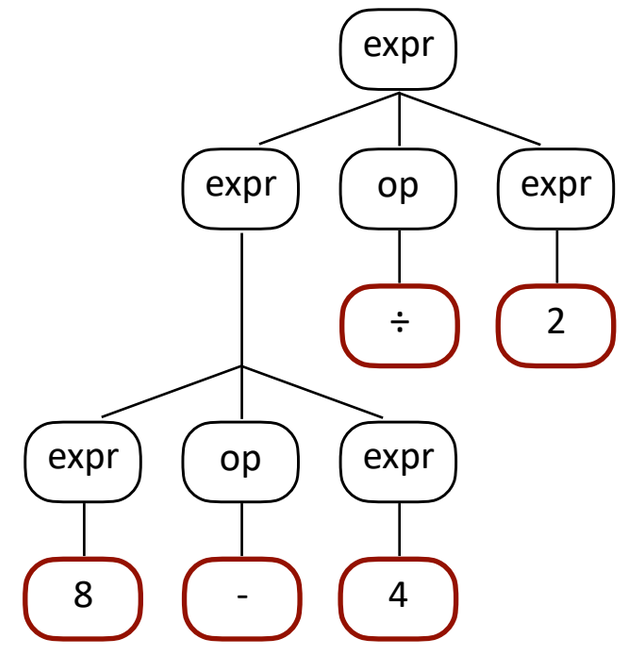
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
[num,8]  -  [num,4]  ÷  <expr>
[num,8]  -  [num,4]  ÷  [num,2]
```



Ambiguous Context-Free Grammars

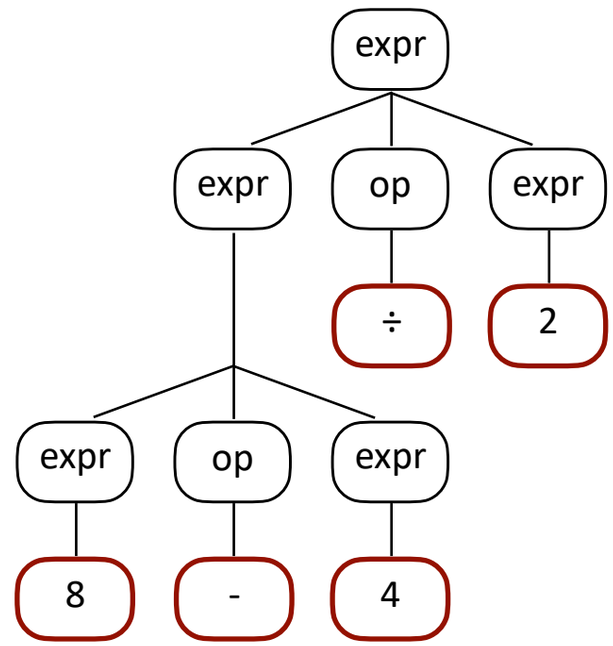
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
[num,8]  -  [num,4]  ÷  <expr>
[num,8]  -  [num,4]  ÷  [num,2]
```



A depth-first in-order traversal of a CST where we print the leaf nodes as we go will yield 8 - 4 ÷ 2.

The shape of the CST determines the grouping.

Ambiguous Context-Free Grammars

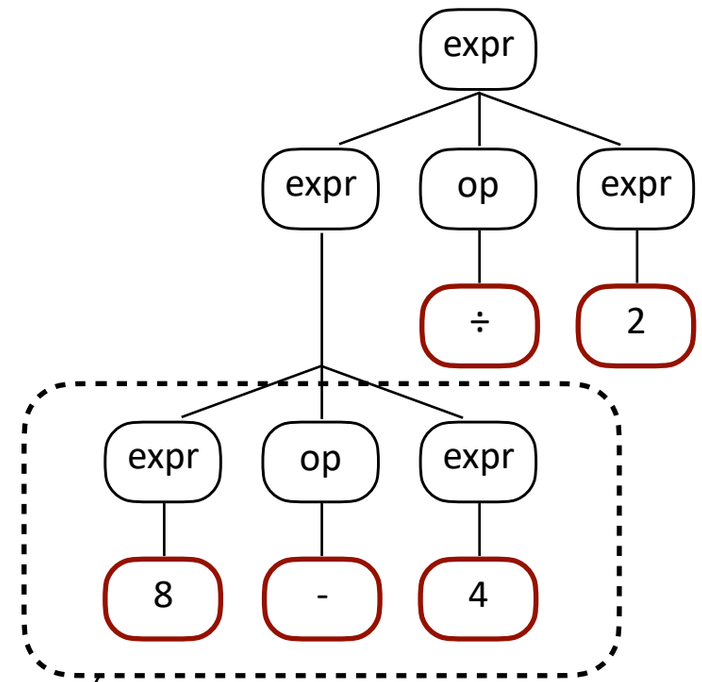
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
[num,8]  -  [num,4]  ÷  <expr>
[num,8]  -  [num,4]  ÷  [num,2]
```



Because of the shape of the tree: (8 - 4) ÷ 2

Ambiguous Context-Free Grammars

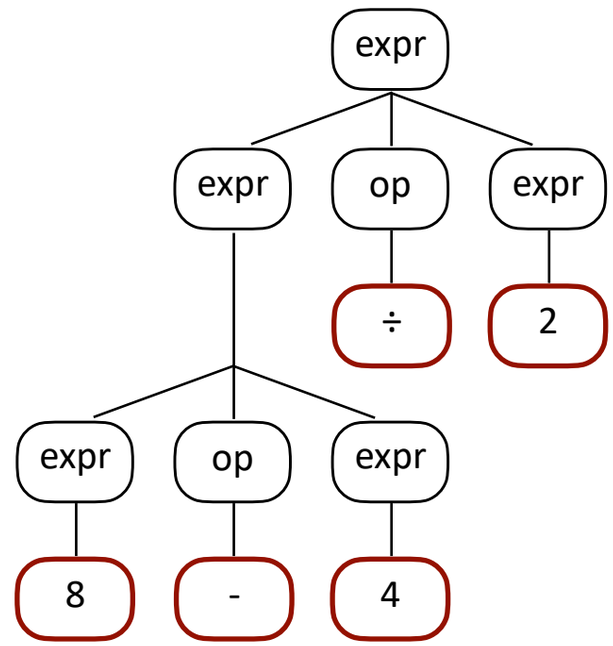
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
[num,8]<op> <expr> <op> <expr>
[num,8]  -  <expr> <op> <expr>
[num,8]  -  [num,4]<op> <expr>
[num,8]  -  [num,4]  ÷  <expr>
[num,8]  -  [num,4]  ÷  [num,2]
```



Because of the shape of the tree: (8 - 4) ÷ 2 = 2, which is wrong.
We broke math.

Ambiguous Context-Free Grammars

A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

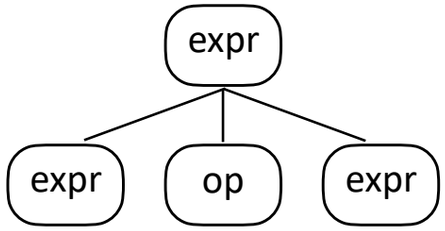
<expr>

Ambiguous Context-Free Grammars

A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

input tokens: `8 - 4 ÷ 2`



Right-most derivation

<expr>

<expr> <op> <expr>

Ambiguous Context-Free Grammars

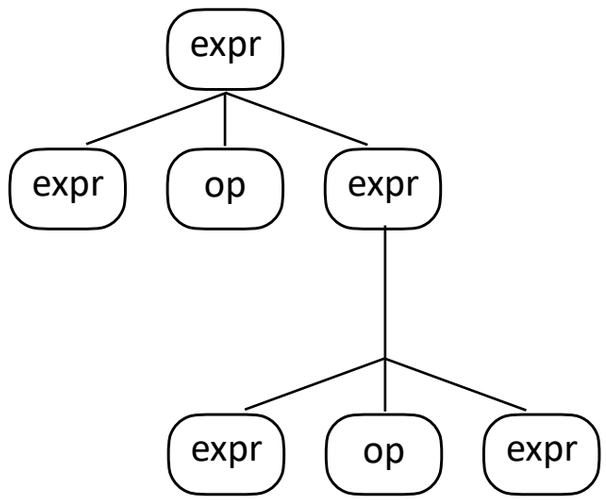
A less cooperative grammar

```
1.  <expr> ::== <expr> <op> <expr>
2.         ::== num
3.  <op>   ::== ÷
4.         ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>



Ambiguous Context-Free Grammars

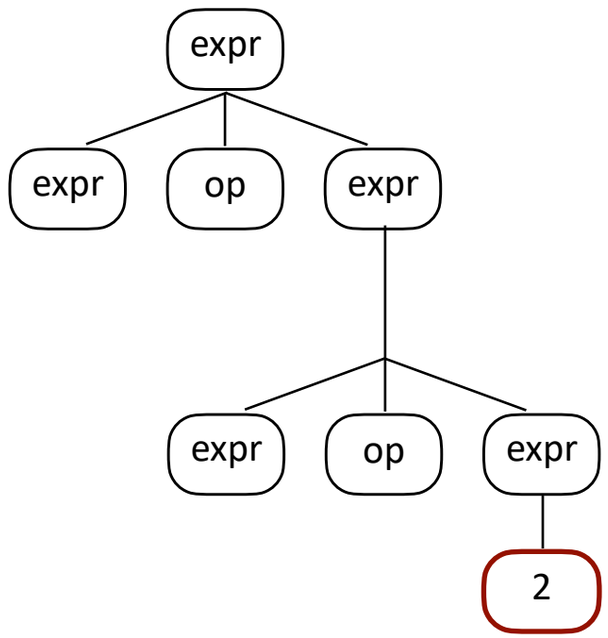
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
```



Ambiguous Context-Free Grammars

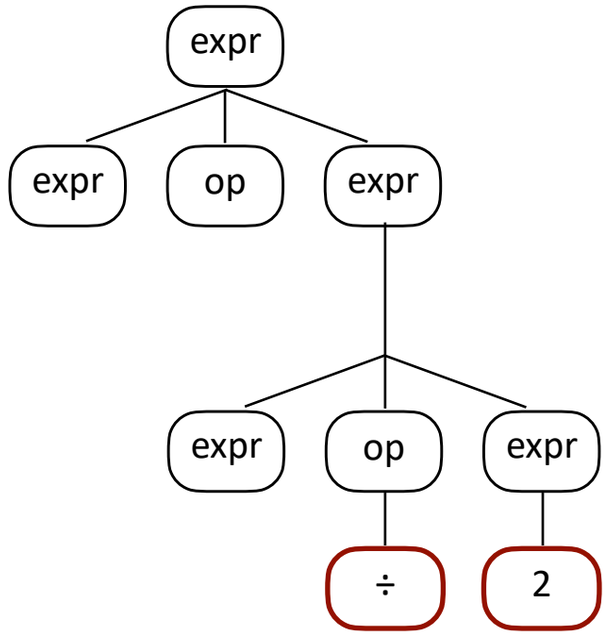
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
```



Ambiguous Context-Free Grammars

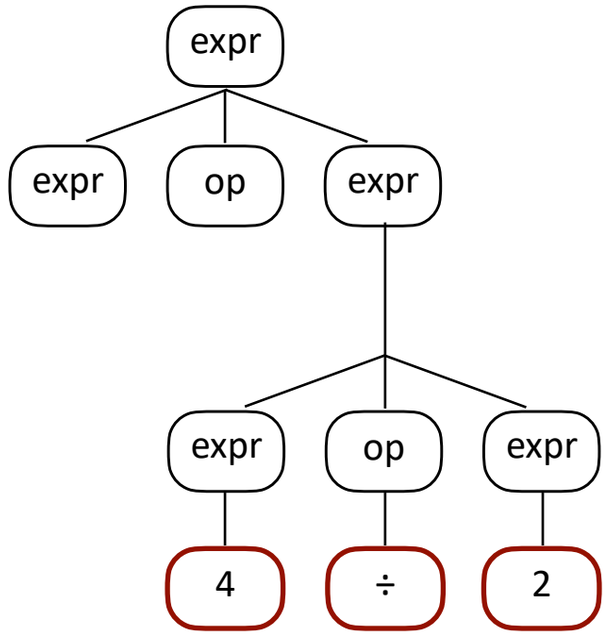
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
```



Ambiguous Context-Free Grammars

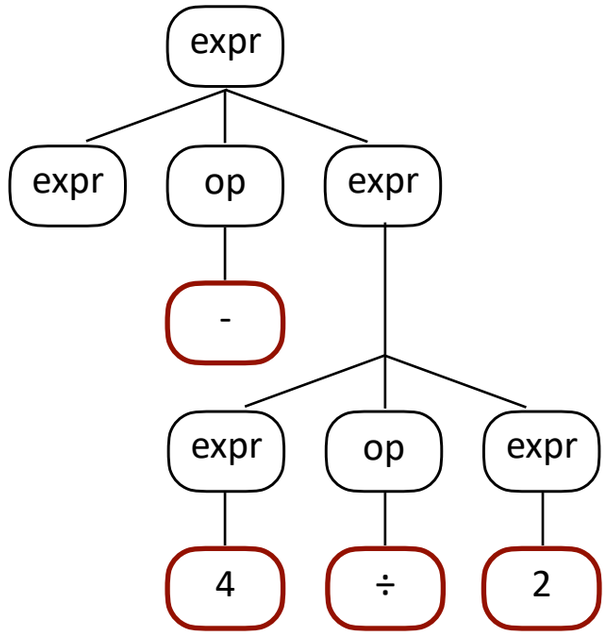
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
<expr>   -  [num,4]  ÷   [num,2]
```



Ambiguous Context-Free Grammars

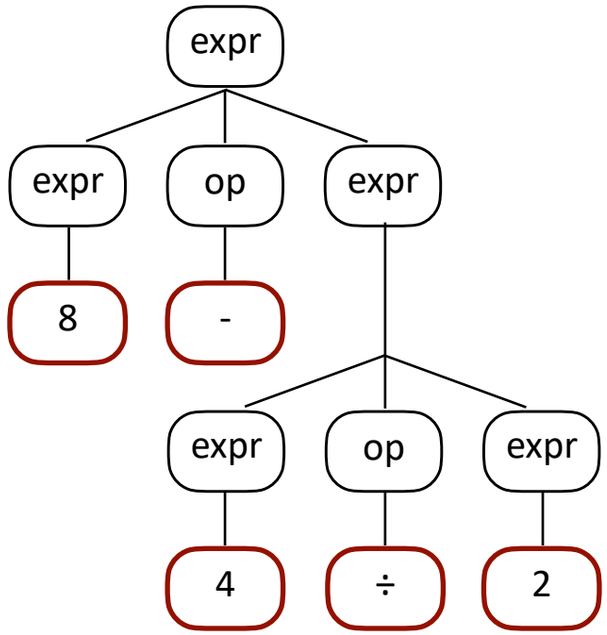
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
<expr>   -  [num,4]  ÷   [num,2]
[num,8]  -  [num,4]  ÷   [num,2]
```



Ambiguous Context-Free Grammars

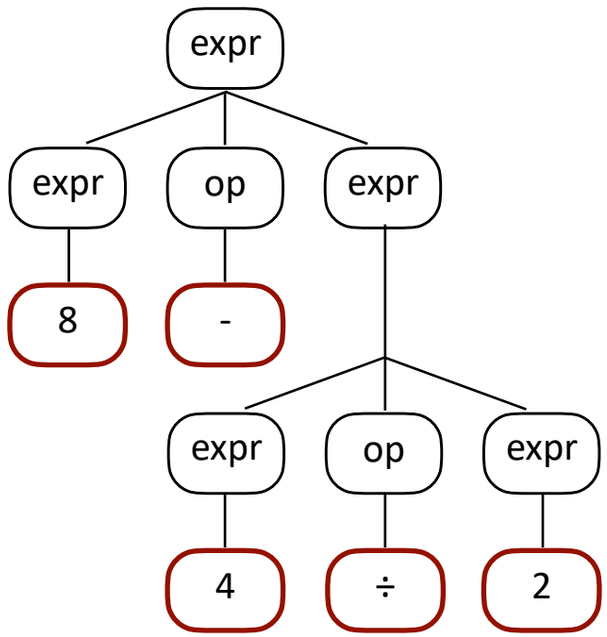
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
<expr>   -  [num,4]  ÷   [num,2]
[num,8]  -  [num,4]  ÷   [num,2]
```



A depth-first in-order traversal of a CST where we print the leaf nodes as we go will yield 8 - 4 ÷ 2.

The shape of the CST determines the grouping.

Ambiguous Context-Free Grammars

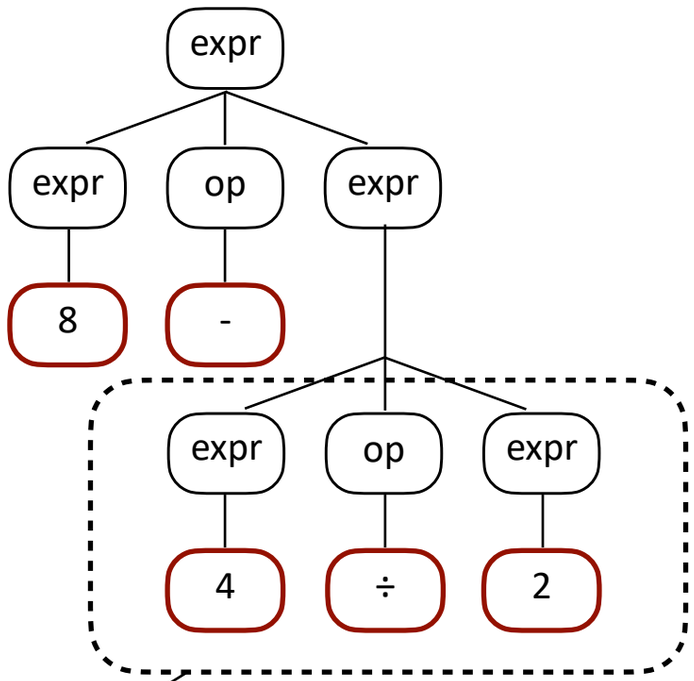
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
<expr>   -  [num,4]  ÷   [num,2]
[num,8]  -  [num,4]  ÷   [num,2]
```



Because of the shape of the tree: 8 - (4 ÷ 2)

Ambiguous Context-Free Grammars

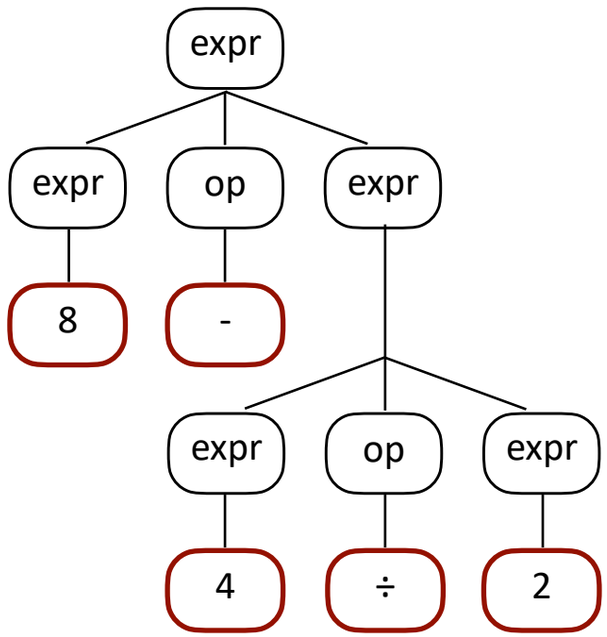
A less cooperative grammar

```
1.  <expr>  ::== <expr> <op> <expr>
2.          ::== num
3.  <op>    ::== ÷
4.          ::== -
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> <op> <expr>
<expr> <op> <expr> <op> <expr>
<expr> <op> <expr> <op> [num,2]
<expr> <op> <expr>   ÷   [num,2]
<expr> <op> [num,4]  ÷   [num,2]
<expr>   -  [num,4]  ÷   [num,2]
[num,8]  -  [num,4]  ÷   [num,2]
```



Because of the shape of the tree: 8 - (4 ÷ 2) = 6, which is correct. Yay, math!

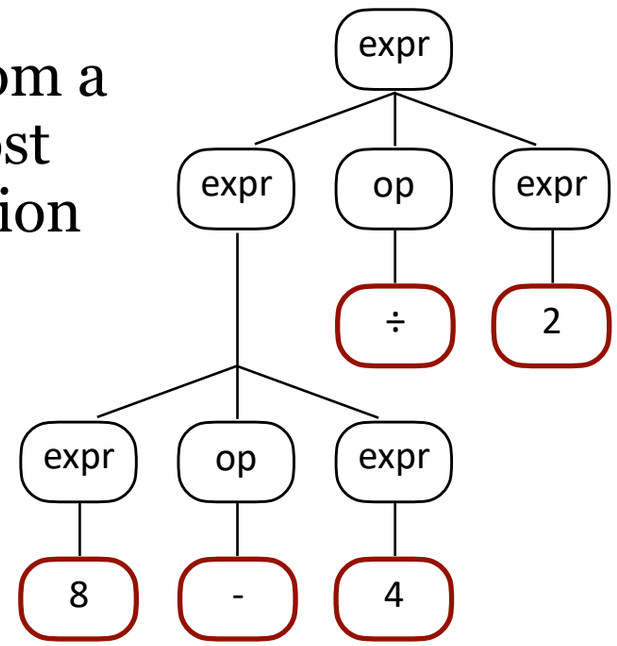
Ambiguous Context-Free Grammars

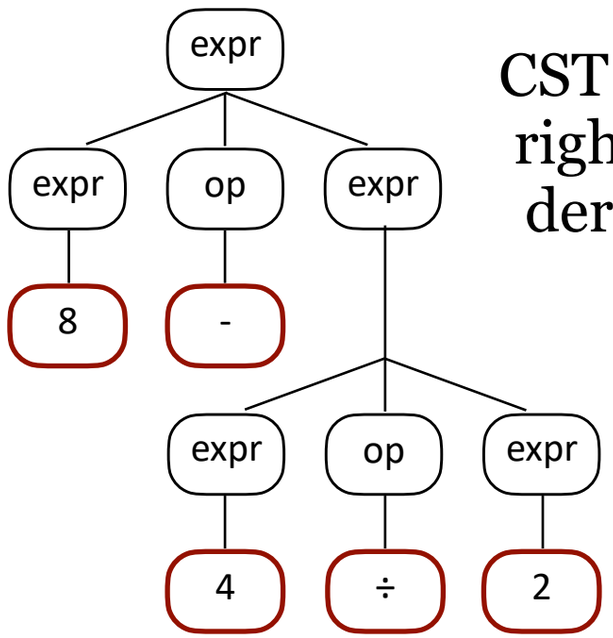
A less cooperative grammar

```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: `8 - 4 ÷ 2`

CST from a left-most derivation





CST from a right-most derivation

This is what happens with ambiguous grammars. Different derivations can lead to different parse trees, which leads to different meanings of the program, which leads to suffering.

~~Ambiguous~~ Context-Free Grammars

A more cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

Context-Free Grammars

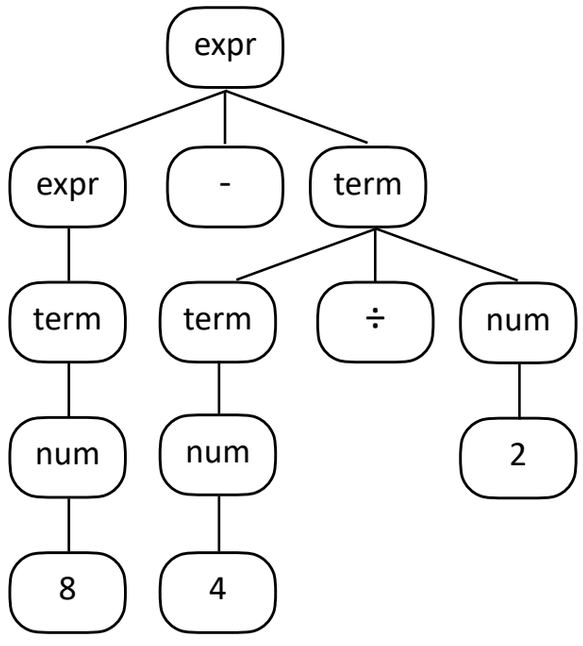
A more cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> - <term>
<term> - <term>
<num>  - <term>
  8    - <term>
  8    - <term> ÷ <num>
  8    - <num>  ÷ <num>
  8    -   4    ÷ <num>
  8    -   4    ÷   2
```



Context-Free Grammars

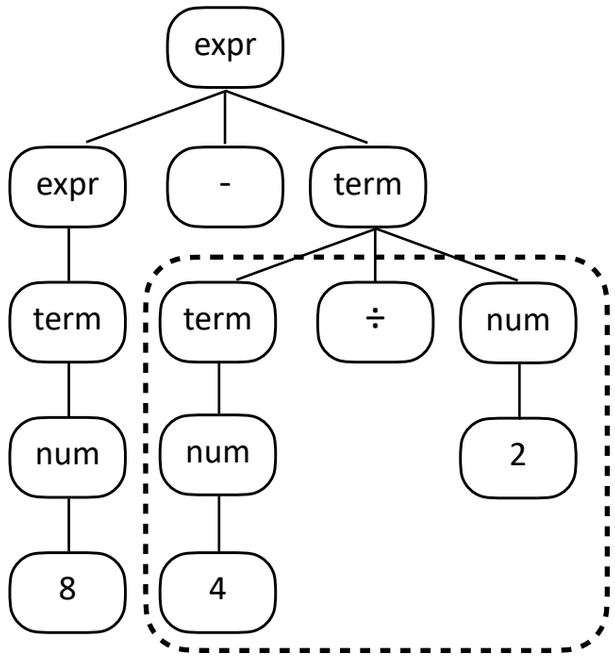
A more cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> - <term>
<term> - <term>
<num>  - <term>
  8    - <term>
  8    - <term> ÷ <num>
  8    - <num>  ÷ <num>
  8    -   4    ÷ <num>
  8    -   4    ÷   2
```



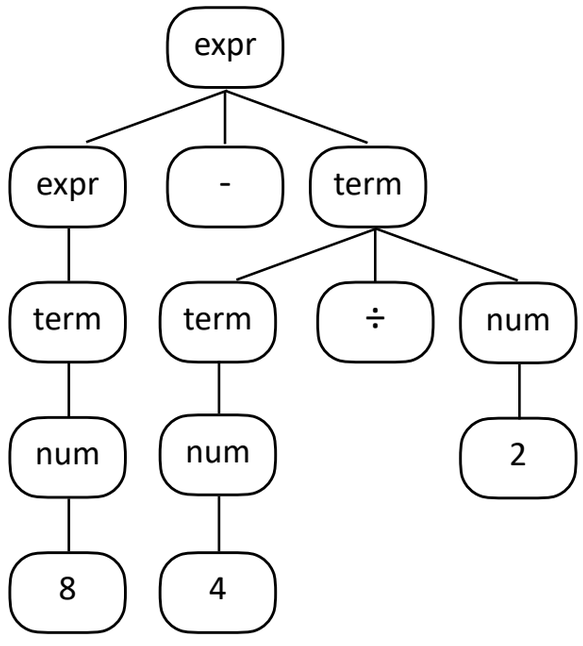
output: `8 - (4 ÷ 2)`

Context-Free Grammars

A more cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`



Right-most derivation

```
<expr>
<expr> - <term>
<expr> - <term> ÷ <num>
<expr> - <term> ÷   2
<expr> - <term> ÷   2
<expr> - <num>  ÷   2
<expr> -   4    ÷   2
<expr> -   4    ÷   2
<term> -   4    ÷   2
<num>  -   4    ÷   2
  8    -   4    ÷   2
```

Context-Free Grammars

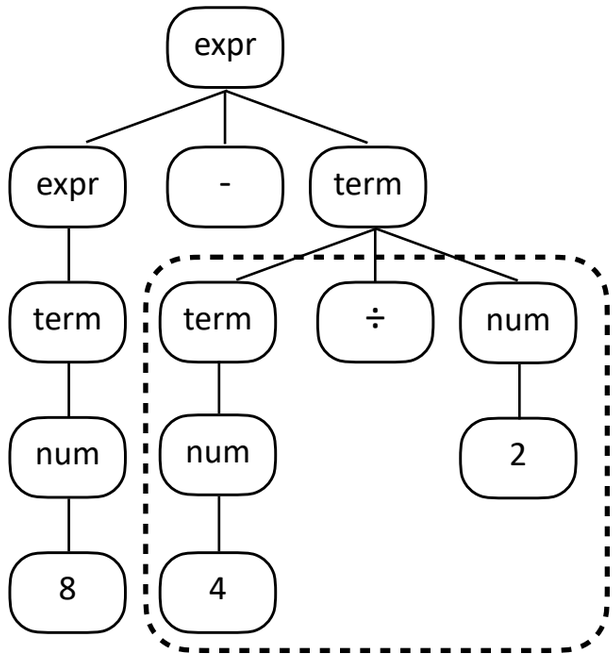
A more cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`

Right-most derivation

```
<expr>
<expr> - <term>
<expr> - <term> ÷ <num>
<expr> - <term> ÷   2
<expr> - <term> ÷   2
<expr> - <num>  ÷   2
<expr> -   4    ÷   2
<expr> -   4    ÷   2
<term> -   4    ÷   2
<num>  -   4    ÷   2
  8    -   4    ÷   2
```



output: `8 - (4 ÷ 2)`

Context-Free Grammars

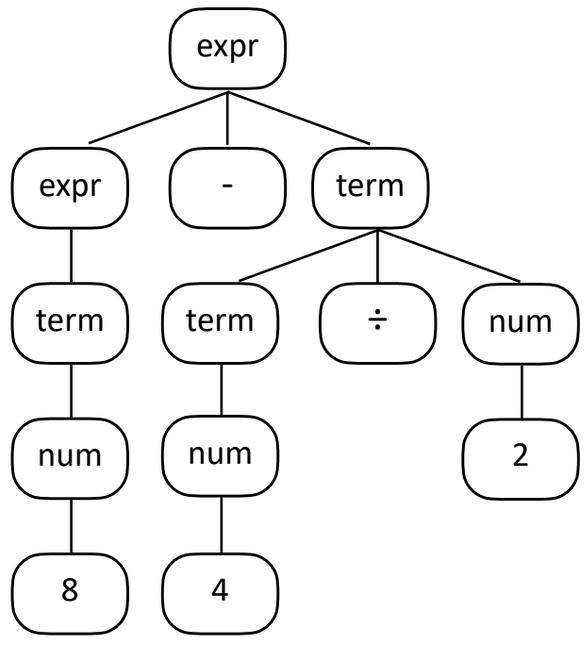
A very cooperative grammar

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`

Left-most derivation

```
<expr>
<expr> - <term>
<term> - <term>
<num>  - <term>
  8    - <term>
  8    - <term> ÷ <num>
  8    - <num>  ÷ <num>
  8    -   4    ÷ <num>
  8    -   4    ÷   2
```



Right-most derivation

```
<expr>
<expr> - <term>
<expr> - <term> ÷ <num>
<expr> - <term> ÷   2
<expr> - <term> ÷   2
<expr> - <num>  ÷   2
<expr> -   4    ÷   2
<expr> -   4    ÷   2
<term> -   4    ÷   2
<num>  -   4    ÷   2
  8    -   4    ÷   2
```

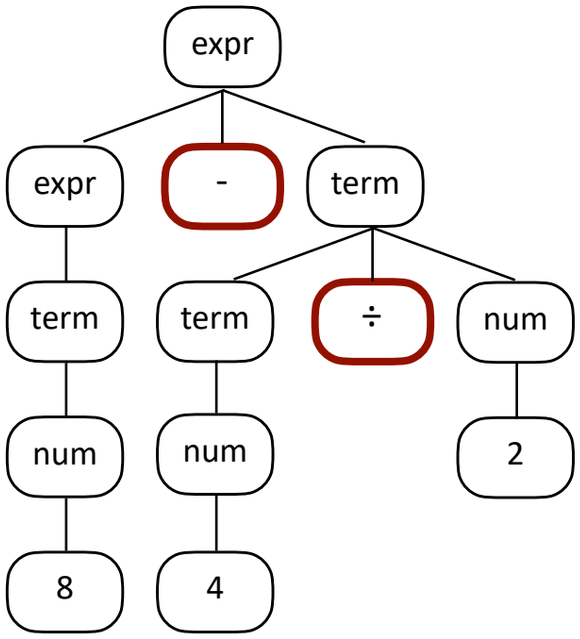
Same CST because
the grammar is unambiguous.

Context-Free Grammars

A note about operator precedence.

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`



Note:
The **higher** the operator precedence, the **lower** it goes in the syntax tree.

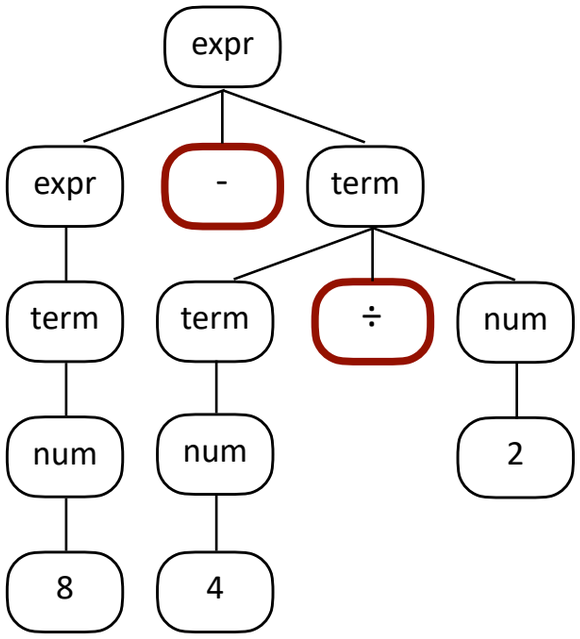
Why is this?

Context-Free Grammars

A note about operator precedence.

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`



Note:
The **higher** the operator precedence, the **lower** it goes in the syntax tree.

This is because of the grammar.

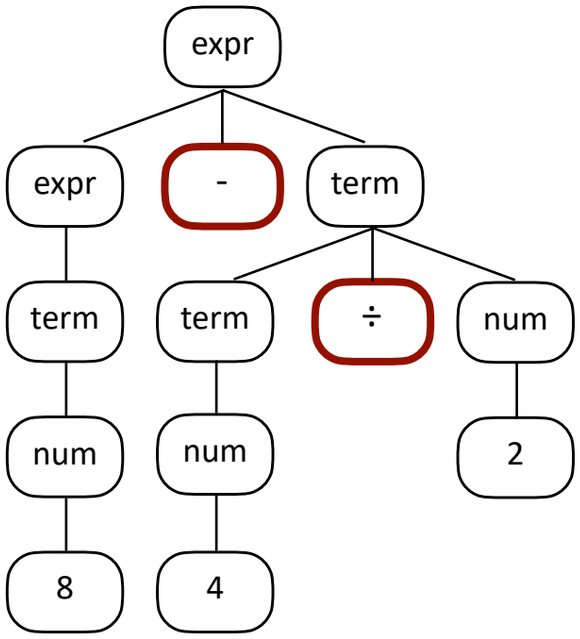
In this example, if there is subtraction, it has to be processed before division because we can only divide **<term>**s so we'll have to go through production #1 to get there, thus placing minus higher in the derivation — and therefore higher in the syntax tree — than division.

Context-Free Grammars

A note about operator precedence.

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`



Note:
The **higher** the operator precedence, the **lower** it goes in the syntax tree.

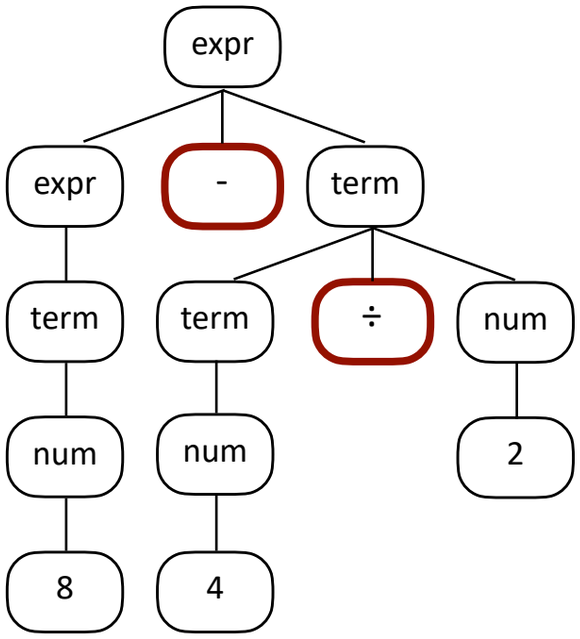
Why do we want this?

Context-Free Grammars

A note about operator precedence.

```
1. <expr> ::== <expr> - <term>
2.        ::== <term>
3. <term> ::== <term> ÷ <num>
4.        ::== <num>
5. <num>  ::== 0|1|2 ··· |9
```

input tokens: `8 - 4 ÷ 2`



Note:
The **higher** the operator precedence, the **lower** it goes in the syntax tree.

We want this because we're going to process our CSTs with depth-first in-order traversals. Since it's **depth-first**, we'll get to and process operators **lower** on the tree before the we get to and process operators higher up the tree.

Non-LL(1) Context-Free Grammars

Consider this grammar

```
1.  <goal>   → <expr>
2.  <expr>   → <expr> + <term>
3.           → <expr> - <term>
4.           → <term>
5.  <term>   → <term> * <factor>
6.           → <term> ÷ <factor>
7.           → <factor>
8.  <factor> → num
9.           → id
```

Issues?

Non-LL(1) Context-Free Grammars

Consider this grammar

```
1.  <goal>   → <expr>
2.  <expr>   → <expr> + <term>
3.           → <expr> - <term>
4.           → <term>
5.  <term>   → <term> * <factor>
6.           → <term> ÷ <factor>
7.           → <factor>
8.  <factor> → num
9.           → id
```

Issues!

- It's ambiguous. (Lots of First set overlap.)
- It's left-recursive.

FIRST(goal) = {**num**, **id**}
FIRST(expr) = {**num**, **id**}
FIRST(term) = {**num**, **id**}
FIRST(factor) = {**num**, **id**}

Let's tweak the grammar to remove left recursion.

Non-LL(1) Context-Free Grammars

Removing Left Recursion

```
1.  <goal>   → <expr>
2.  <expr>   → <expr> + <term>
3.           → <expr> - <term>
4.           → <term>
5.  <term>   → <term> * <factor>
6.           → <term> ÷ <factor>
7.           → <factor>
8.  <factor> → num
9.           → id
```

```
1.  <goal>   → <expr>
2.  <expr>   → <term> + <expr>
3.           → <term> - <expr>
4.           → <term>
5.  <term>   → <factor> * <term>
6.           → <factor> ÷ <term>
7.           → <factor>
8.  <factor> → num
9.           → id
```

Issues?

- Well… now it's right-recursive, which is nice (and parseable).
- It's also right-associative, which is less nice, but not the problem we're dealing with at the moment.

Progress, but still not LL(1).

Non-LL(1) Context-Free Grammars

Removed Left Recursion

```
1.  <goal>   → <expr>
2.  <expr>   → <term> + <expr>
3.           → <term> - <expr>
4.           → <term>
5.  <term>   → <factor> * <term>
6.           → <factor> ÷ <term>
7.           → <factor>
8.  <factor> → num
9.           → id
```

FIRST(goal) = {**num**, **id**}
FIRST(expr) = {**num**, **id**}
FIRST(term) = {**num**, **id**}
FIRST(factor) = {**num**, **id**}

Issues:

- Still not LL(1) because of First set overlap.
- For **<expr>** how can we choose among productions 2, 3, and 4? We need to look "past" the **<term>** to get to the next token.
- And for **<term>**? Same thing.

We can tweak the grammar again to fix this.

Non-LL(1) Context-Free Grammars

Left Factoring

```
1.  <goal>    → <expr>
2.  <expr>    → <term> + <expr>
3.            → <term> - <expr>
4.            → <term>
5.  <term>    → <factor> * <term>
6.            → <factor> ÷ <term>
7.            → <factor>
8.  <factor>  → num
9.            → id
```

```
<expr>   → <term> <expr'>
<expr'>  → + <expr>
         → - <expr>
         → ε
```

Resolve the issue by “factoring out” the non-terminals causing overlap in the First set, and introducing (distinct) terminals into the new First set.

Non-LL(1) Context-Free Grammars

Left Factoring

```
1.  <goal>   → <expr>
2.  <expr>   → <term> <expr'>
3.  <expr'>  → + <expr>
4.           → - <expr>
5.           → ε
6.  <term>     → <factor> * <term>
7.             → <factor> ÷ <term>
8.             → <factor>
9.  <factor> → num
10.          → id
```

```
<term>   → <factor> <term'>
<term'>  → * <term>
         → ÷ <term>
         → ε
```

Resolve the issue by "factoring out" the non-terminals causing overlap in the First set, and introducing (distinct) terminals into the new First set.

~~Non-~~LL(1) Context-Free Grammars

Left Factored

```
1.  <goal>  → <expr>
2.  <expr>  → <term> <expr'>
3.  <expr'> → + <expr>
4.          → - <expr>
5.          → ε
6.  <term>  → <factor> <term'>
7.  <term'> → * <term>
8.          → ÷ <term>
9.          → ε
10. <factor> → num
11.          → id
```

This new, slightly larger, grammar is LL(1).

- It's not left-recursive.
- All First sets within a Production are distinct.

But what if we didn't want to tweak the grammar…?