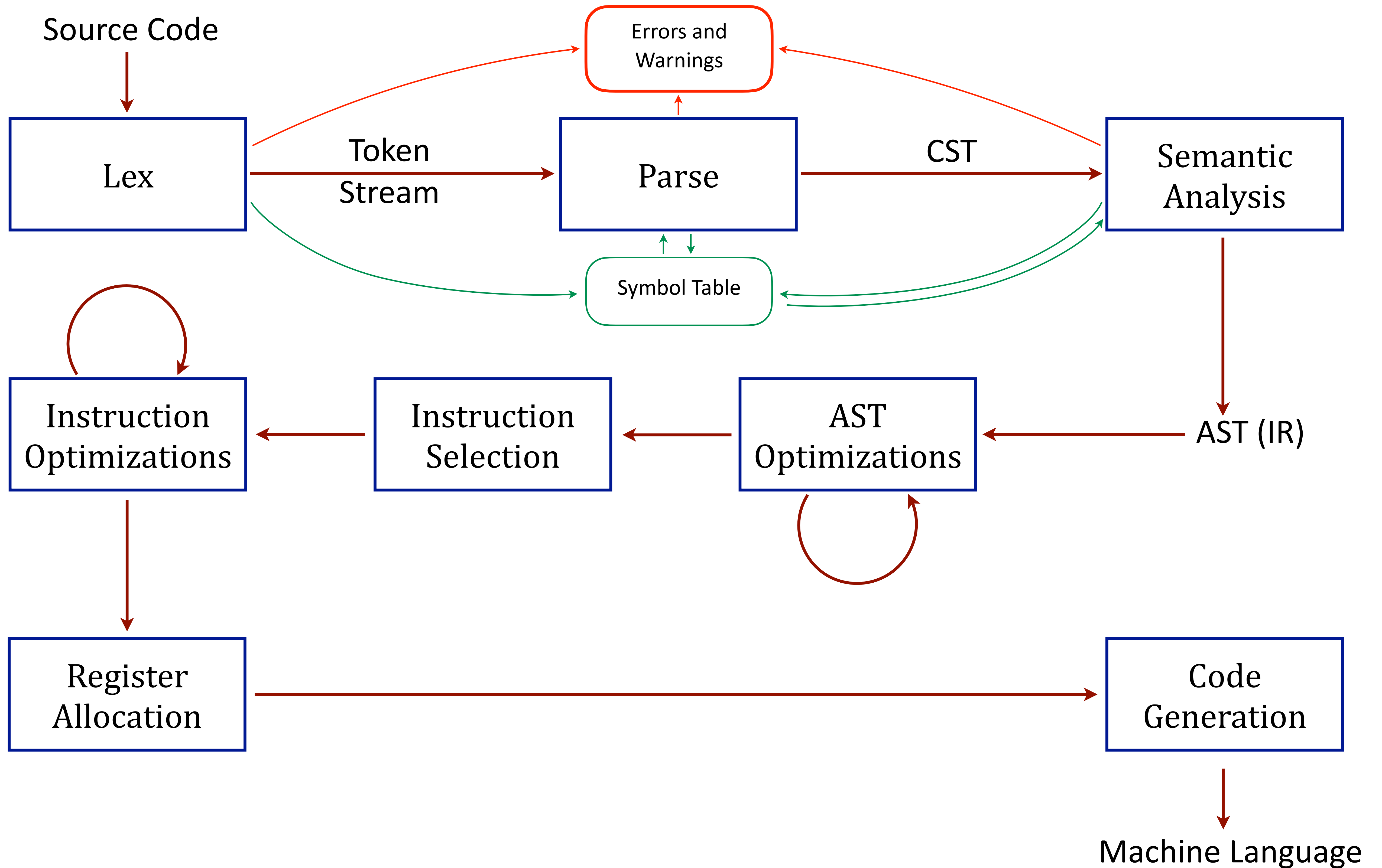
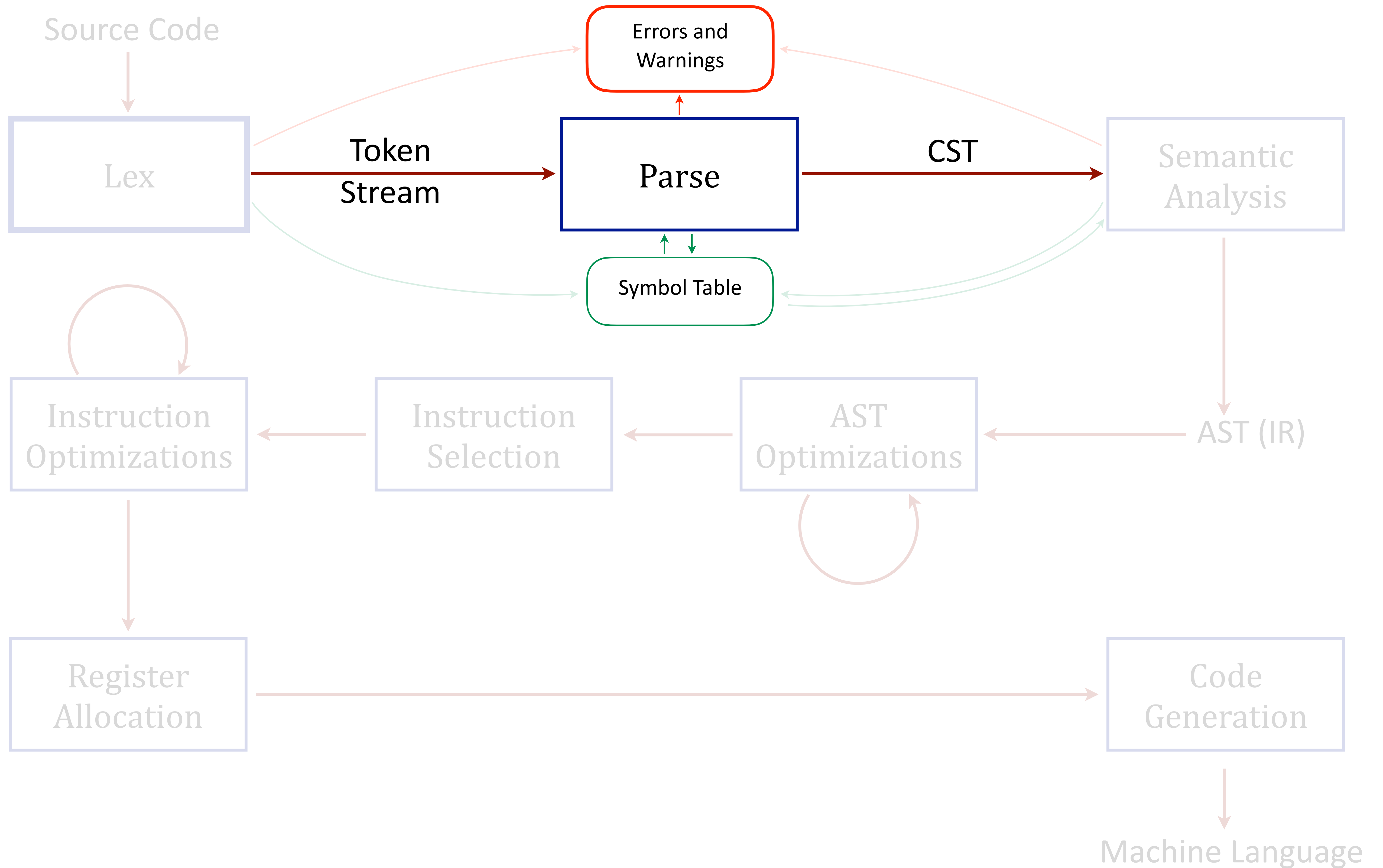
Parsing

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

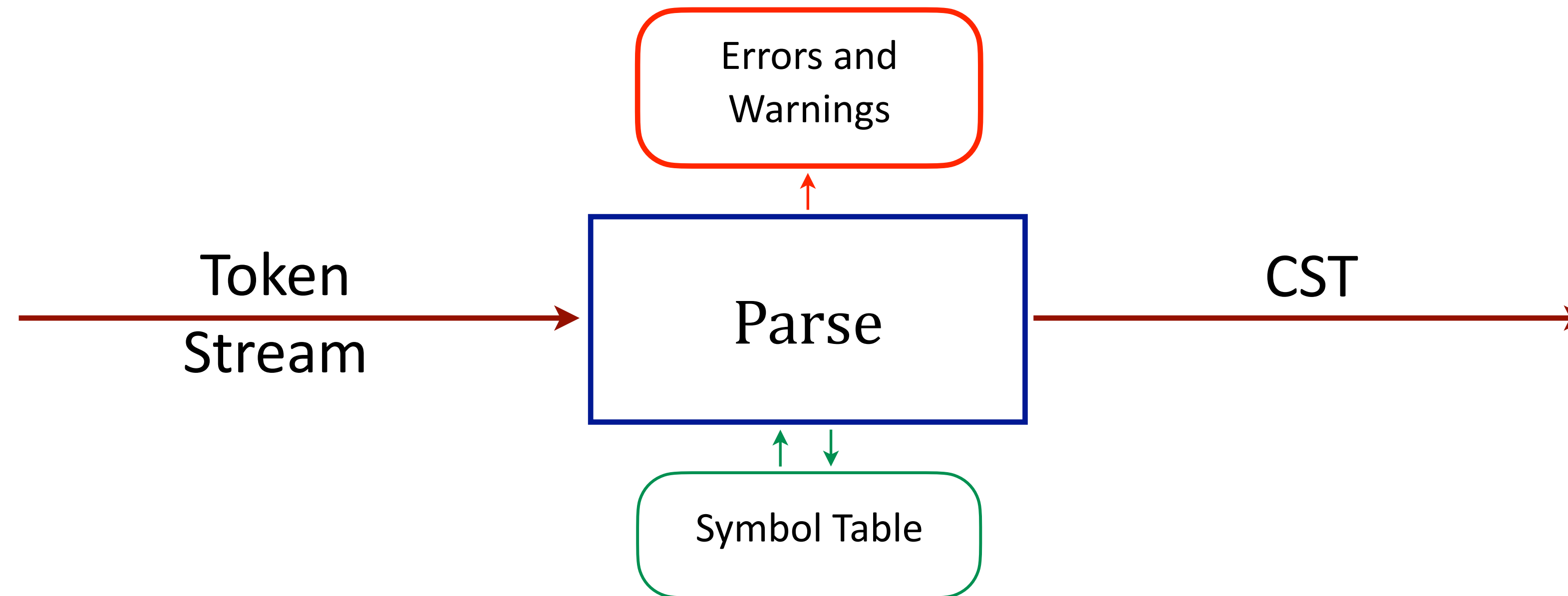
Compiler — High Level View



Compiler — High Level View



High Level View: Parse



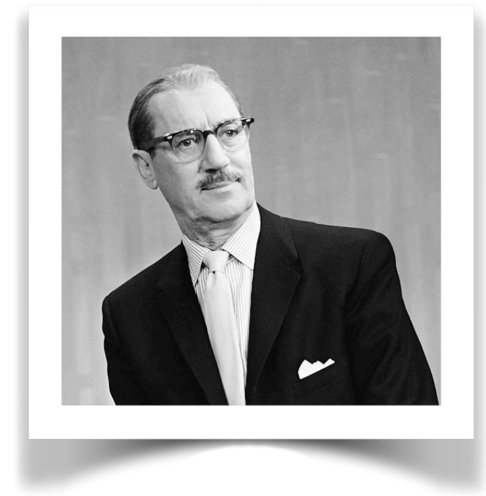
Parse

- Recognize context-free syntax
- Recognize variables and type
- Report meaningful errors and warnings
- Produce a parse tree (aka: Concrete Syntax Tree) for Semantic Analysis.

- Focus on syntax

Parsing Sentences

Time flies like an arrow.



Fruit flies like an apple.

How can you tell these two sentences apart?
They are very similar in structure, but not in meaning.

Parsing Sentences

preposition prepositional object

Time flies like an arrow.

noun verb indefinite article

Fruit flies like an apple.

How can you tell these two sentences apart?
They are very similar in structure, but not in meaning.

We can look at the parts of speech as we read/parse them.

Parsing Sentences

preposition prepositional object
Time flies like an arrow.
noun verb indefinite article

indefinite article
Fruit flies like an apple.
adjective noun verb direct object

How can you tell these two sentences apart?
They are very similar in structure, but not in meaning.

We can look at the parts of speech as we read/parse them, even if we might not be certain until we reach the end. This is known as *syntactic ambiguity*, or a *garden path sentence*.

Parsing Sentences

Garden Path Sentences

That thing where you begin parsing it one way, but have to **back track** at some later point and try a different parse.

- The old man the boat.
- The government plans to raise taxes were defeated.
- I convinced her children are cute.

We will use concepts similar to parts of speech in parsing programming languages.

We will also need to be wary of garden path sentences and embrace backtracking from time to time.

Parsing Sentences

Parsing focuses on **syntax**, not meaning.

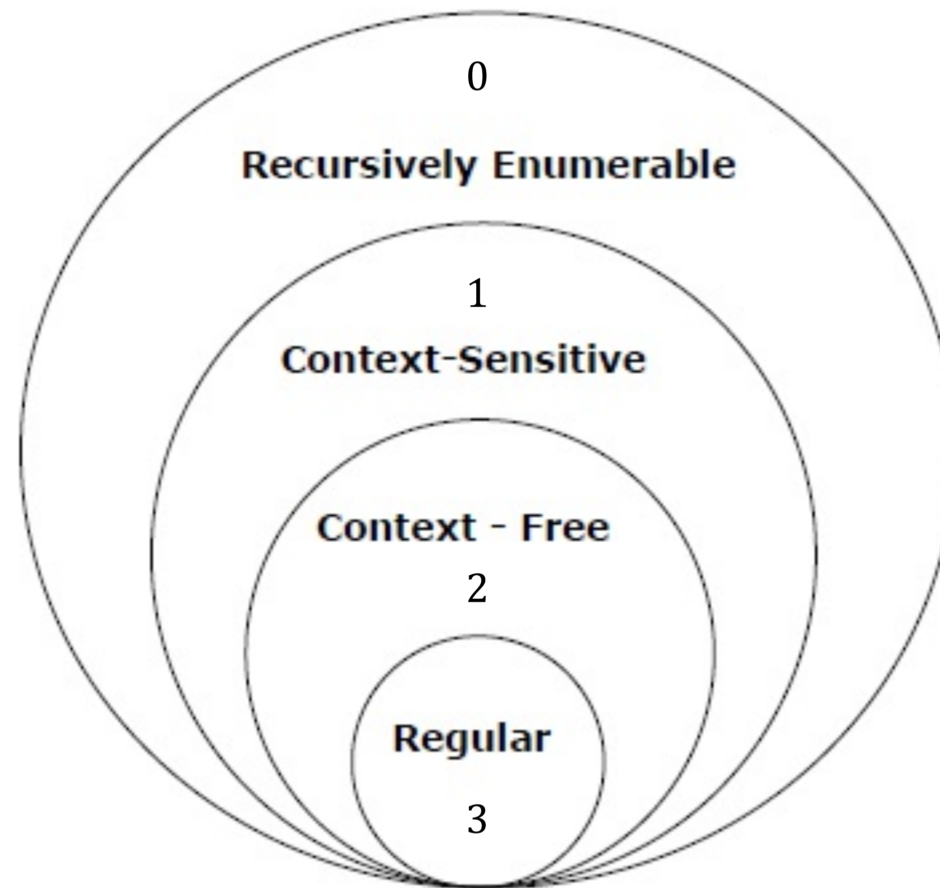
We can have syntactically correct sentences that don't make sense.*

Colorless green ideas sleep furiously.

Structure and syntax (words and sentences) are independent from semantics (meaning).

* A poet may disagree.

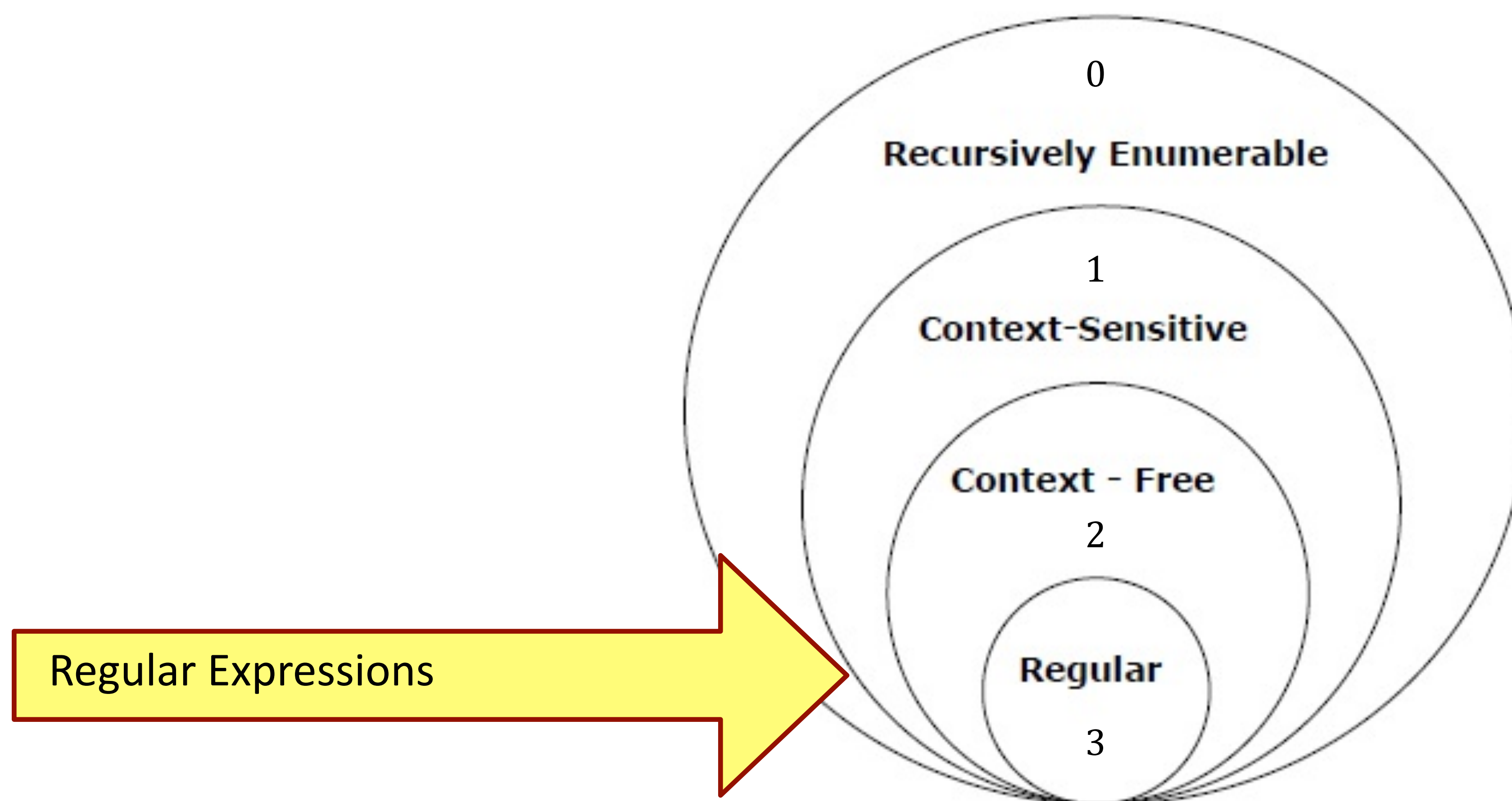
Chomsky Hierarchy



In the 1950s Noam Chomsky classified the syntactic complexity of languages into four categories.

Each “higher” category contains the prior categories within it.

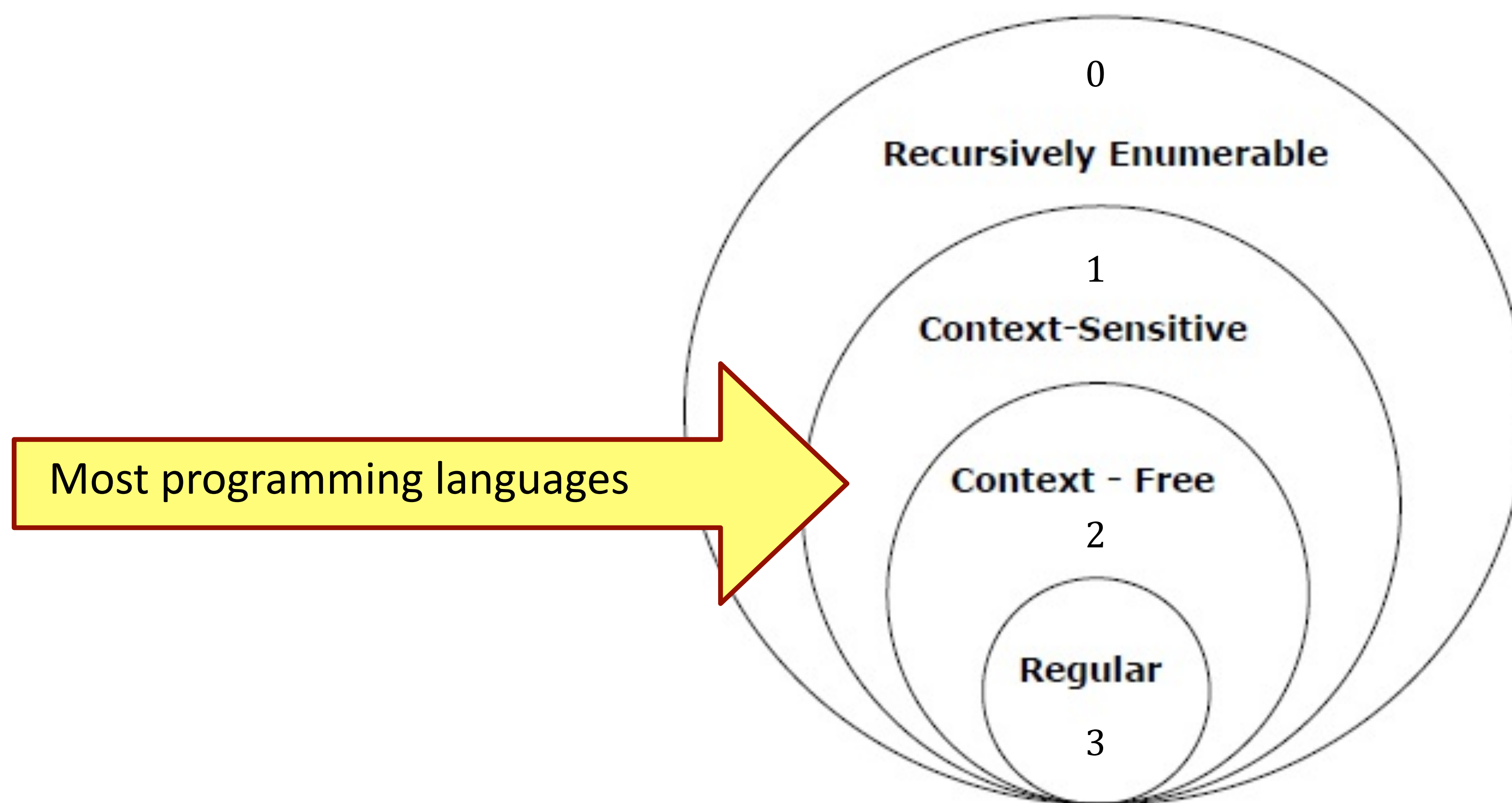
Chomsky Hierarchy



Regular Sets, or Regular Expressions as we know and love them, are the least expressive languages. They are still incredibly useful for things like pattern matching, of course.

```
(a|b)*abb
```

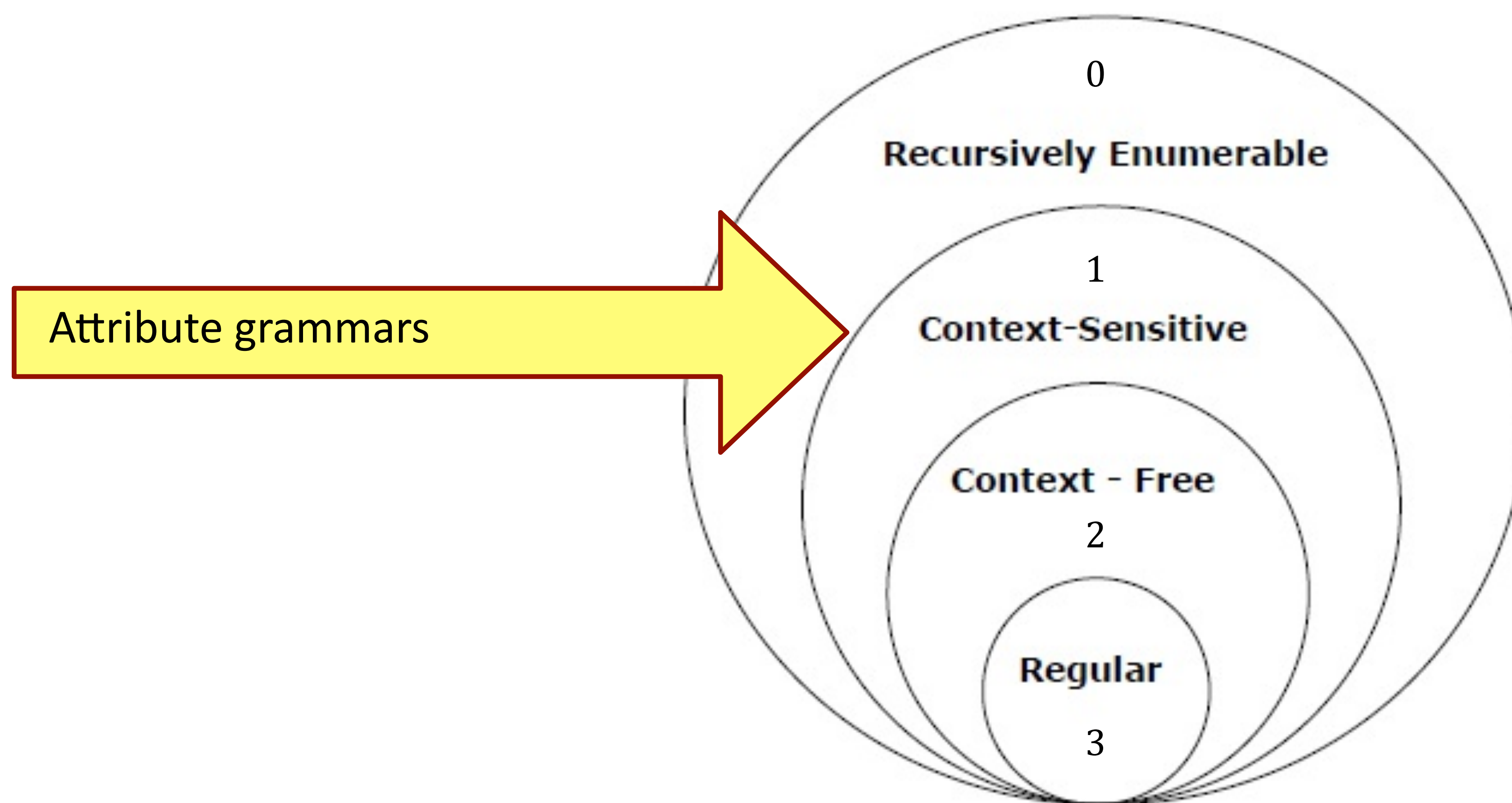
Chomsky Hierarchy



Context-free grammars (CFG) are more expressive than Regular languages, and we're going to spend a lot of time on them. Most programming languages are of this type.

```
1. <sheep noise> → baa
```

Chomsky Hierarchy



Context-sensitive grammars are more powerful than CFGs and can enforce things like type matching and sequential requirements.

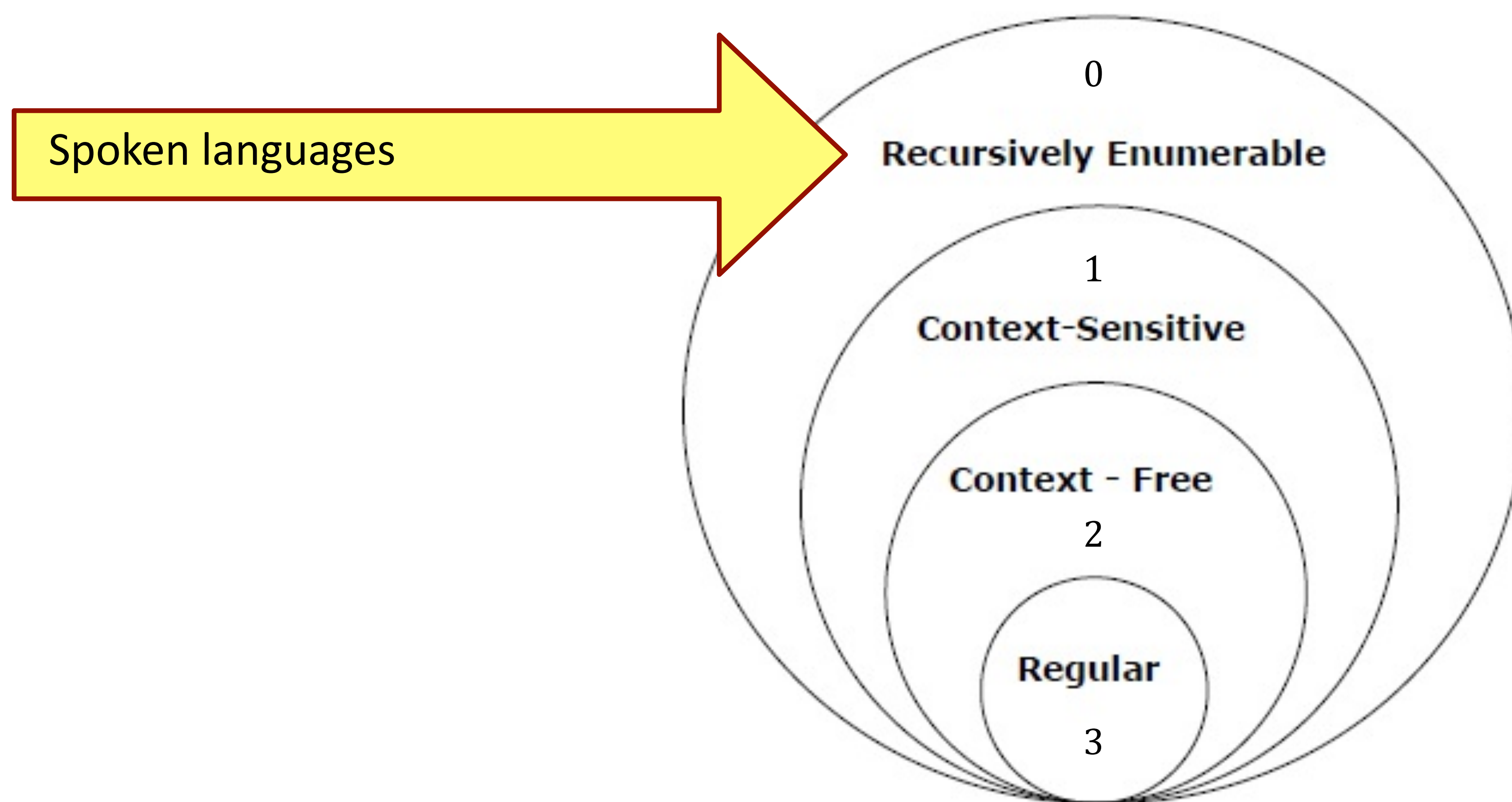
<letter sequence> ::= <a sequence> <b sequence> <c sequence>

<asequence> ::= **a** | <a sequence>**a**

<bsequence> ::= **b** | <bsequence> **b**

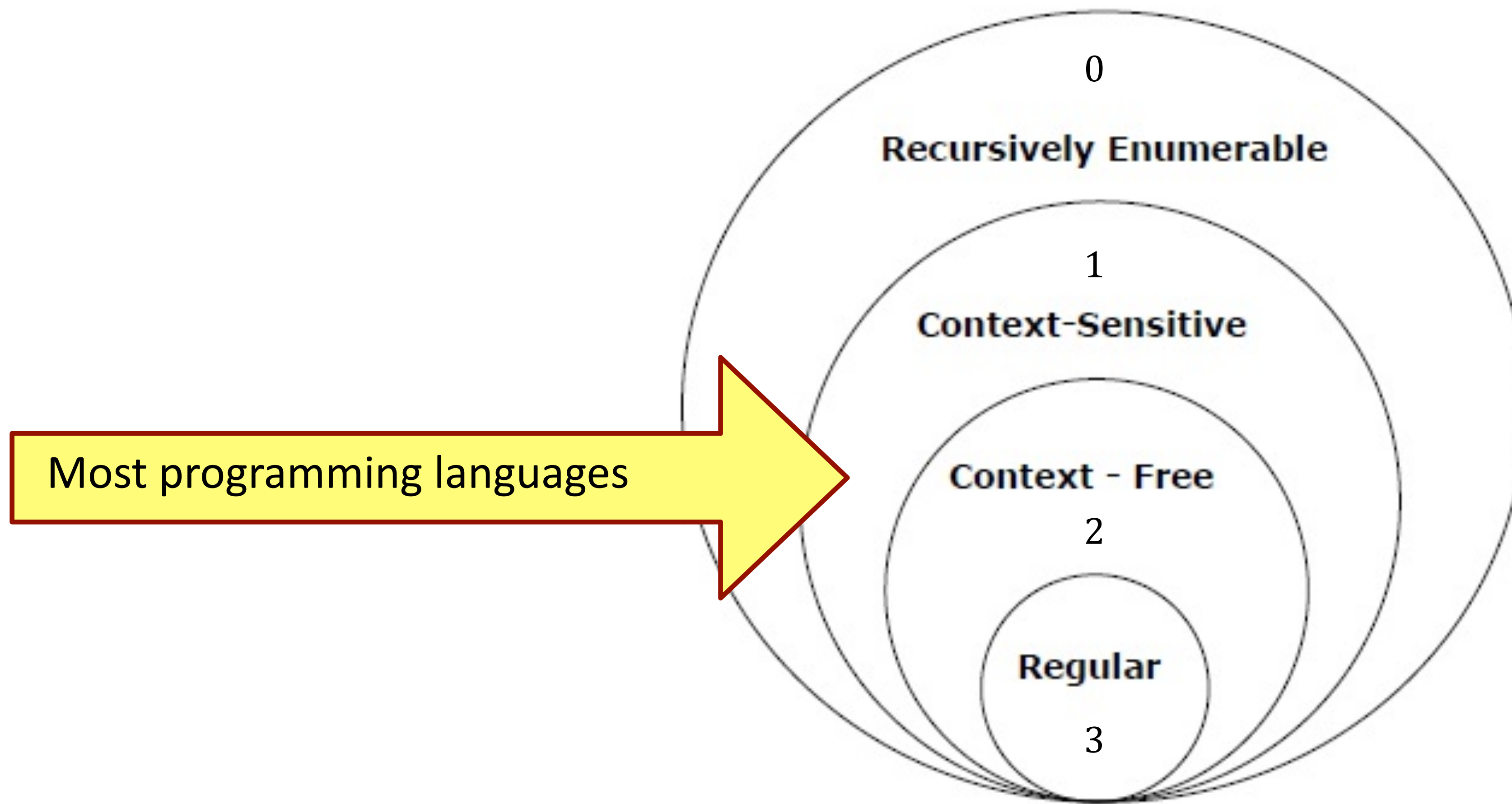
<csequence> ::= **c** | <csequence> **c**

Chomsky Hierarchy



Recursively enumerable grammars are the most powerful and unrestricted; anything goes. (*Difficult* to implement.)

Context-Free Grammars



Context-Free Grammars

Defining a context-free grammar, *G* :

G = (*S*, *N*, *T*, *P*)

where

S is the start symbol

N is a set of non-terminal symbols

T is a set of terminal symbols

P is a set of productions (or *rewrite rules*)

Backus-Naur Form (BNF) is a context-free grammar that's used to define context-free grammars (including itself). It's recursively cool!

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
```

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
```

S is the start symbol.

N is a set of non-terminal symbols.

T is a set of terminal symbols.

P is a set of productions.

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
```

S = <sheep noise>

N = { <sheep noise> }

T = { baa }

P = {1}

If the input is **`baa`** we can prove that it's a valid sentence in our grammar by showing each step of the parse.

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
```

input token: **`baa`**

Parse:
By production #1 we can replace **`<sheep noise>`** with **`baa`**. That consumes the first (and only) token from the input. We've exhausted the input tokens and there are no non-terminals to be replaced with terminals so we're done and have a successful parse.

Easy.

But this only supports very terse sheep. What about **`baa baa`** ?

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
```

input tokens: `baa baa`

Parse:
By production #1 we can replace `<sheep noise>` with `baa`. That consumes the first token from the input. We've **not** exhausted the input tokens but there are no non-terminals to be replaced with terminals so we're done and have an **un**successful parse, indicating an error.

We need another production.

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
2. <sheep noise> → baa <sheep noise>
```

input tokens: **baa baa**

Parse:
By production #2 we can replace **<sheep noise>** with **baa <sheep noise>**. That consumes the first token from the input.
By production #1 we can replace **<sheep noise>** with **baa**.
That consumes the second and last token from the input.
We've exhausted the input tokens and there are no more non-terminals to be replaced with terminals so we're done and have a successful parse.

Context-Free Grammars

Backus-Naur Form (BNF)

Consider a grammar for the noise a sheep makes.

```
1. <sheep noise> → baa
2. <sheep noise> → baa <sheep noise>
```

Recursion!

input tokens: **`baa baa`**

Parse:
By production #2 we can replace **`<sheep noise>`** with **`baa <sheep noise>`**. That consumes the first token from the input.
By production #1 we can replace **`<sheep noise>`** with **`baa`**.
That consumes the second and last token from the input.
We've exhausted the input tokens and there are no more non-terminals to be replaced with terminals so we're done and have a successful parse.

Context-Free Grammars

Carnac Normal Form

A grammar for the sound
made when a sheep explodes.

```
1. <sheep noise> → sis boom baa
```

Context-Free Grammars

Derivations

Consider this grammar from the Dragon book . . .

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

. . . and input tokens: `9 - 5 + 2`

Context-Free Grammars

Derivations

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

```
S. <list>
```

Begin with the start symbol <list>.

list

We start with the start symbol.
Then what?

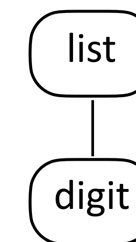
There are three `<list>` productions.
Which should we choose?
Let's try #3.

Context-Free Grammars

Derivations

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

S. <list>	Begin with the start symbol <list>.
3. <digit>	Production #3 : <list> → <digit>



Context-Free Grammars

Derivations

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

```
S. <list>
3. <digit>
4.    9
```

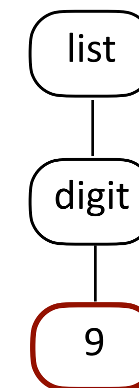
Begin with the start symbol <list>.

Production #3 : <list> → <digit>

Production #4 : <digit> → **9**

Consume **9** from the token stream.

Move the input pointer to the next token in the stream.



Context-Free Grammars

Derivations

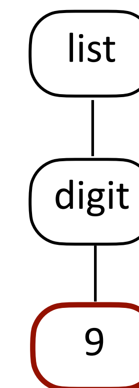
```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

```
S. <list>
3. <digit>
4.    9
```

Begin with the start symbol <list>.

Production #3 : <list> → <digit>

Production #4 : <digit> → **9**



Now what?

Context-Free Grammars

Derivations

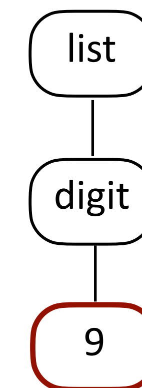
```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

```
S. <list>
3. <digit>
4.    9
```

Begin with the start symbol <list>.

Production #3 : <list> → <digit>

Production #4 : <digit> → **9**



Now what?

There are no more non-terminals to be turned into terminals, but there are still tokens on the input stream.

We must have gone down a wrong (garden?) path.

Backtrack and try a different path.

Context-Free Grammars

Derivations

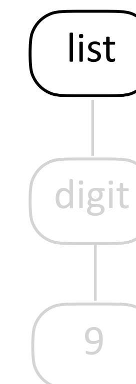
```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```

```
S. <list>
3. <digit>
4.    9
```

Begin with the start symbol <list>.

Production #3 : <list> → <digit>

Production #4 : <digit> → 9



Context-Free Grammars

Derivations

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

```
S. <list>
```

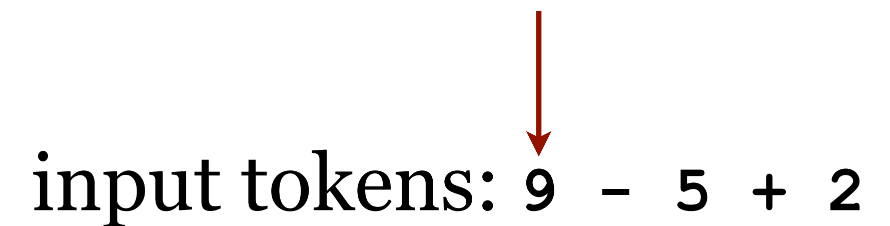
Begin with the start symbol <list>.

list

Context-Free Grammars

Derivations

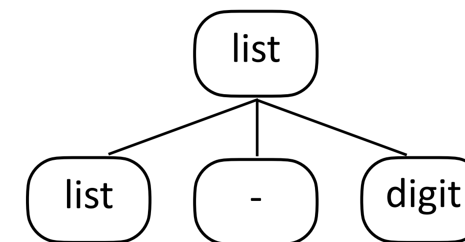
```
1. <list>  ::== <list> + <digit>
2. <list>  ::== <list> - <digit>
3. <list>  ::== <digit>
4. <digit> ::== 0|1|2|3|4|5|6|7|8|9
```




```
S. <list>
2. <list> - <digit>
```

Begin with the start symbol <list>.

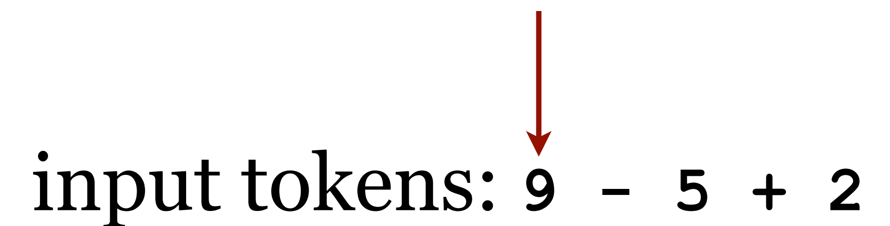
Production #2 : <list> → <list> - <digit>



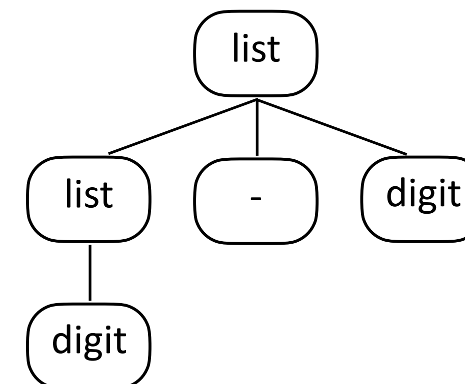
Context-Free Grammars

Derivations

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



S. <list>	Begin with the start symbol <list>.
2. <list> - <digit>	Production #2 : <list> → <list> - <digit>
3. <digit>- <digit>	Production #3 : <list> → <digit>



Context-Free Grammars

Derivations

```
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>
2. <list> - <digit>
3. <digit>- <digit>
4.    9   - <digit>
```

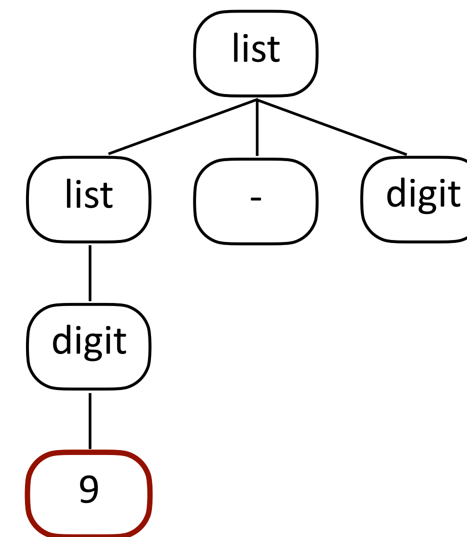
Begin with the start symbol <list>.

Production #2 : <list> → <list> - <digit>

Production #3 : <list> → <digit>

Production #4 : <digit> → **9**

Consume **9** from the token stream.

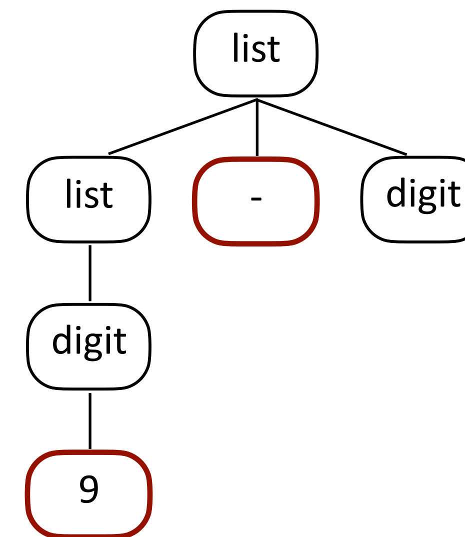


Context-Free Grammars

Derivations

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

S. <list>	Begin with the start symbol <list>.
2. <list> - <digit>	Production #2 : <list> → <list> - <digit>
3. <digit>- <digit>	Production #3 : <list> → <digit>
4. 9 - <digit>	Production #4 : <digit> → **9**
	Consume **9** from the token stream.
	Consume **–** from the token stream.



Context-Free Grammars

Derivations

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

Begin with the start symbol <list>.

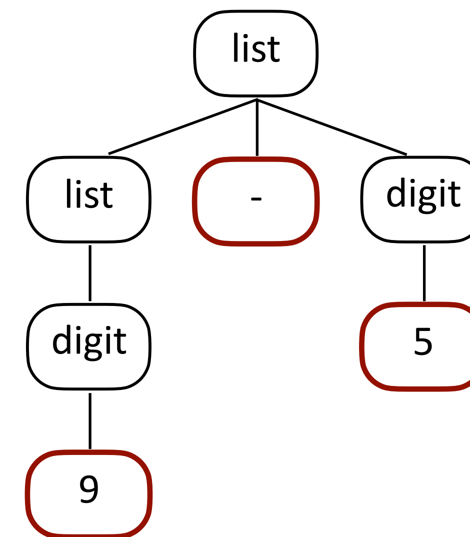
Production #2 : <list> → <list> - <digit>

Production #3 : <list> → <digit>

Production #4 : <digit> → **9**

Production #4 : <digit> → **5**

Consume **5** from the token stream.



Context-Free Grammars

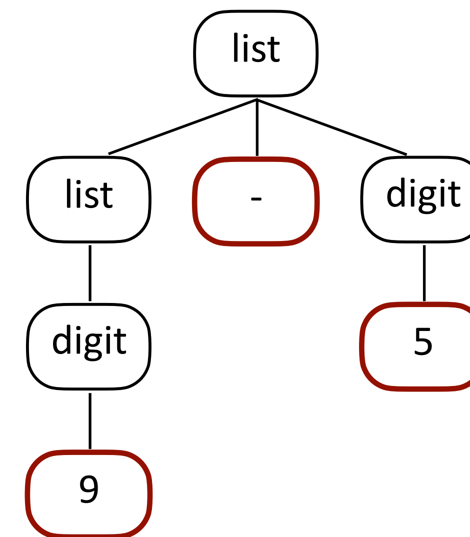
Derivations

```
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: ~~x~~ ~~x~~ ~~x~~ + 2

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

Begin with the start symbol <list>.
Production #2 : <list> → <list> - <digit>
Production #3 : <list> → <digit>
Production #4 : <digit> → **9**
Production #4 : <digit> → **5**



Now what?
Oops. We must have gone down another wrong path.
Backtrack and try again.

Context-Free Grammars

Derivations

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

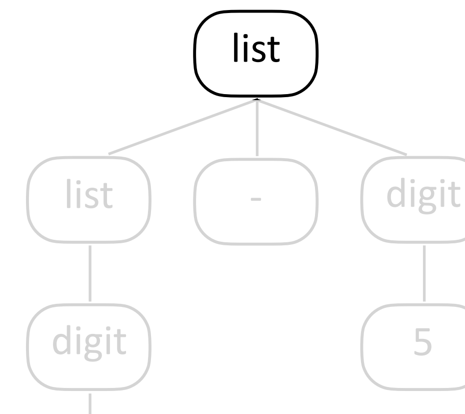
Begin with the start symbol <list>.

Production #2 : <list> → <list> - <digit>

Production #3 : <list> → <digit>

Production #4 : <digit> → 9

Production #4 : <digit> → 5



Context-Free Grammars

Derivations

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

```
S. <list>
```

Begin with the start symbol <list>.

list

Context-Free Grammars

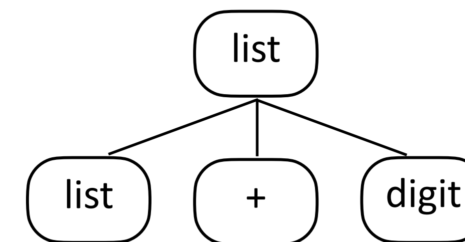
Derivations

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

```
S. <list>
1. <list> + <digit>
```

Begin with the start symbol <list>.

Production #1 : <list> → <list> + <digit>

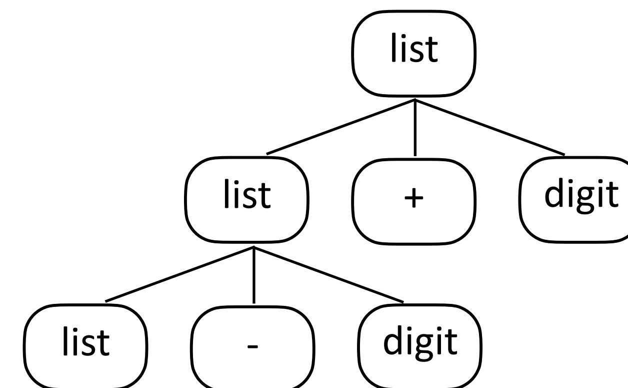


Context-Free Grammars

Derivations

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

```
S. <list>
1. <list> + <digit>
2. <list> - <digit> + <digit>
```

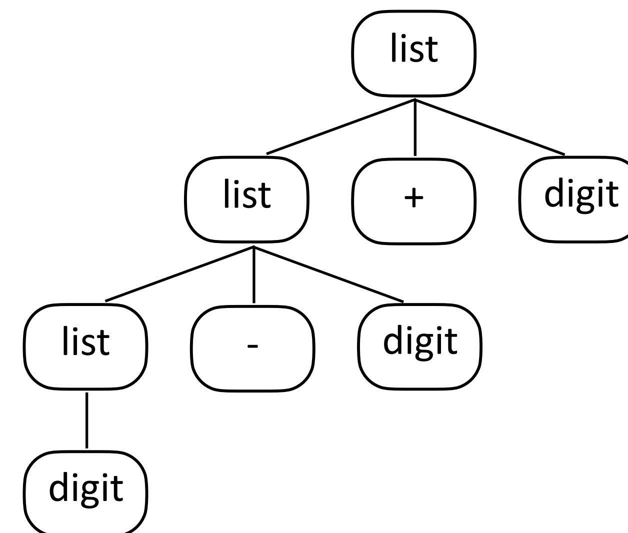


Context-Free Grammars

Derivations

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

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



Context-Free Grammars

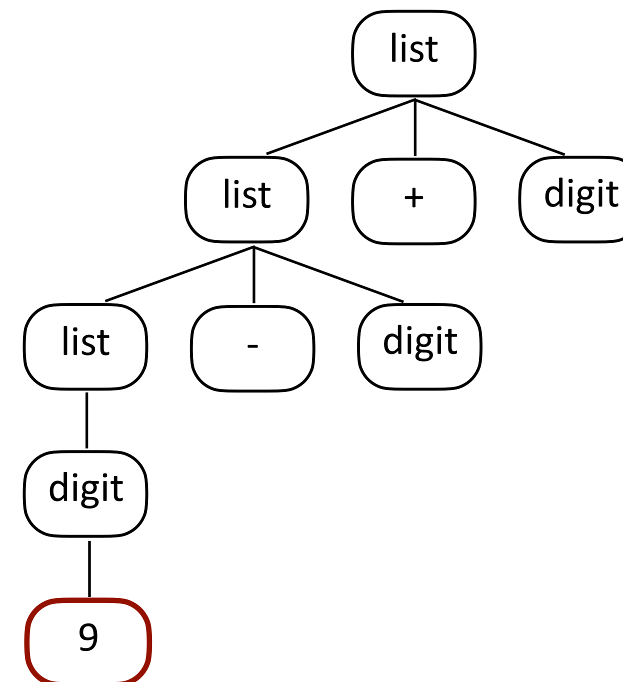
Derivations

```
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>
```

Consume **9** from the token stream.



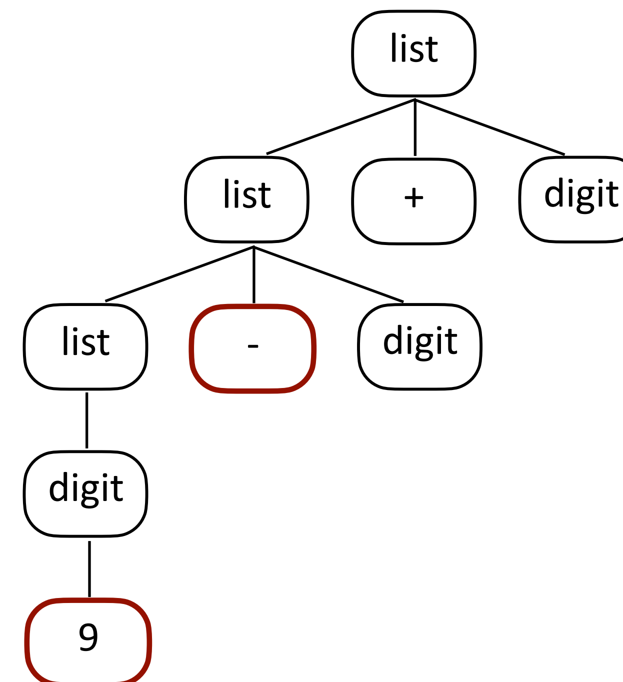
Context-Free Grammars

Derivations

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

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

Consume – from the token stream.



Context-Free Grammars

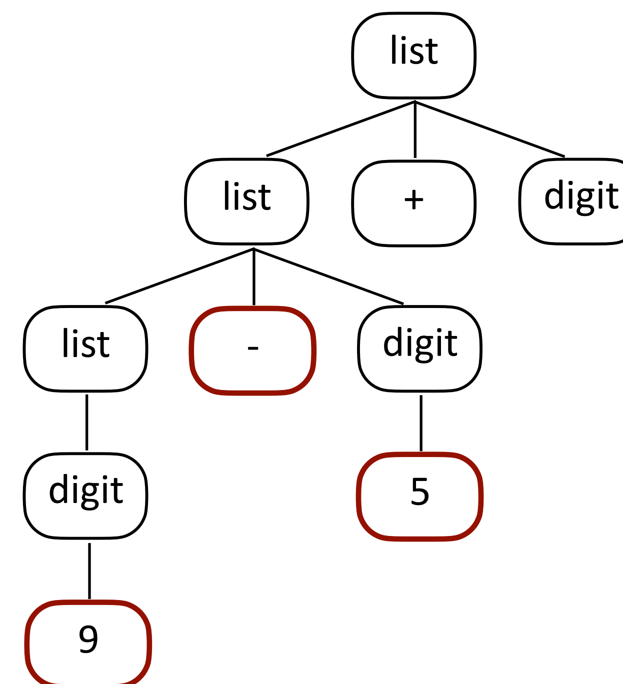
Derivations

```
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: X X X + 2

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

Consume **5** from the token stream.



Context-Free Grammars

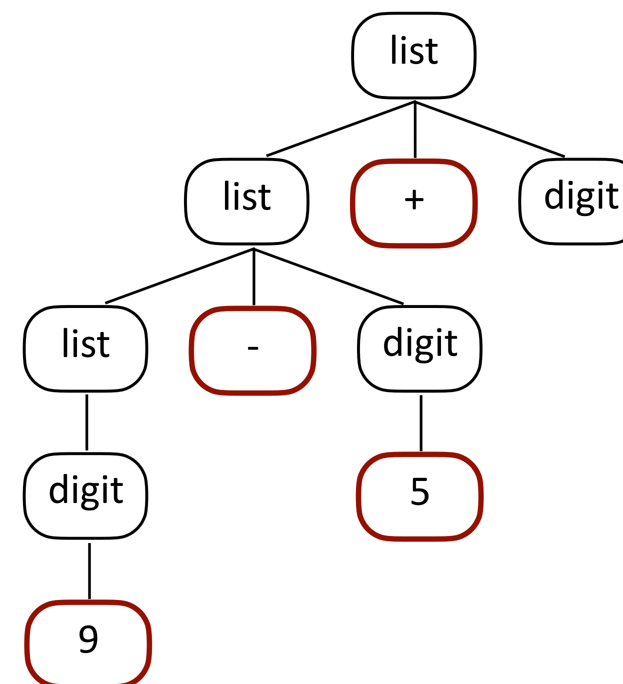
Derivations

```
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: X X X X 2

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

Consume **+** from the token stream.



Context-Free Grammars

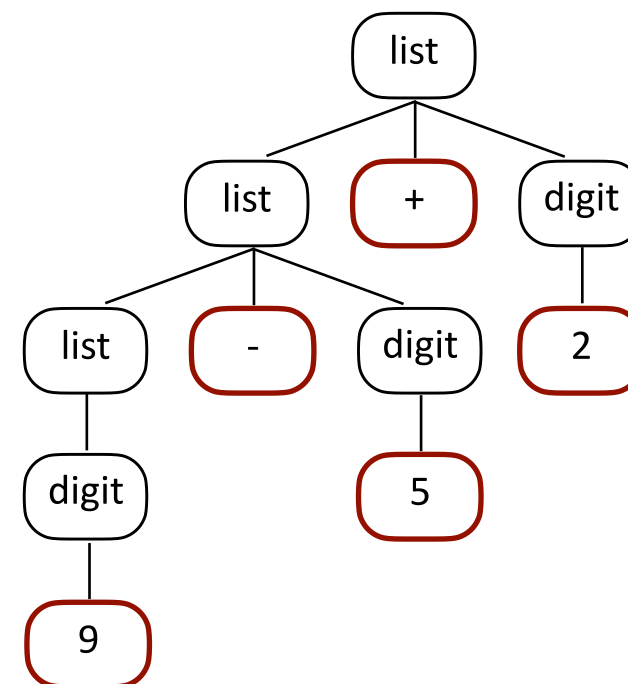
Derivations

```
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:

```
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
```

Consume **2** from the token stream.



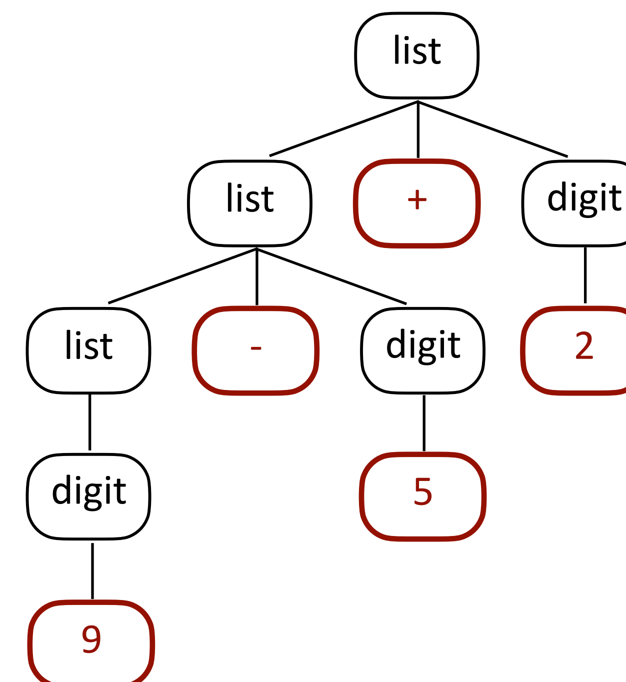
Context-Free Grammars

Derivations

```
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:

```
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.

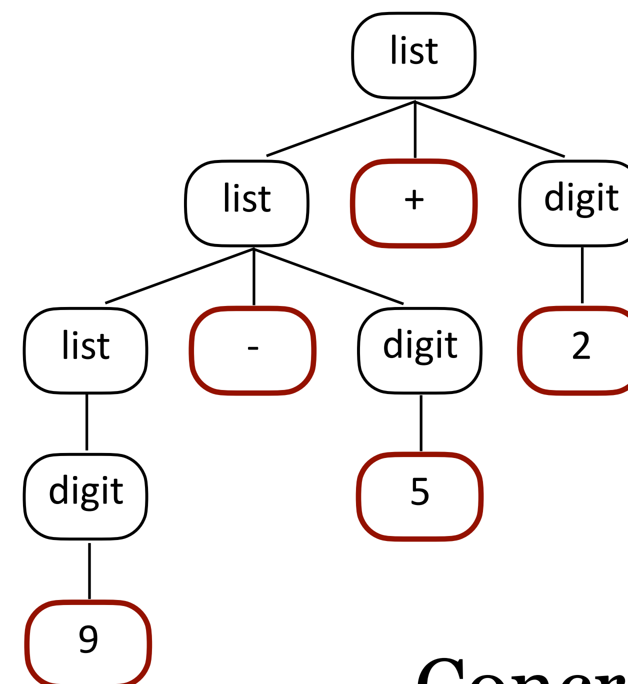
Context-Free Grammars

Left-most Derivation

```
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
```



Concrete Syntax Tree (CST)

This is a **left-most** derivation. Meaning... we processed non-terminals from left to right.

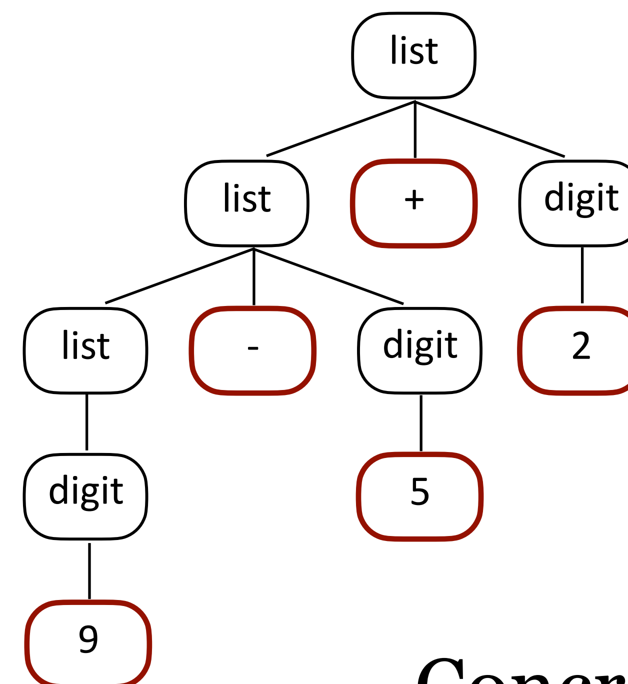
Context-Free Grammars

Left-most Derivation

```
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:

```
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
```



Concrete Syntax Tree (CST)

This is a **left-most** derivation.

Let’s try a right-most derivation.

Context-Free Grammars

Right-most Derivation

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

```
S. <list>
```

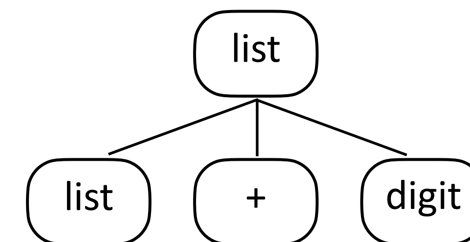
list

Context-Free Grammars

Right-most Derivation

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

```
S. <list>
1. <list>           + <digit>
```



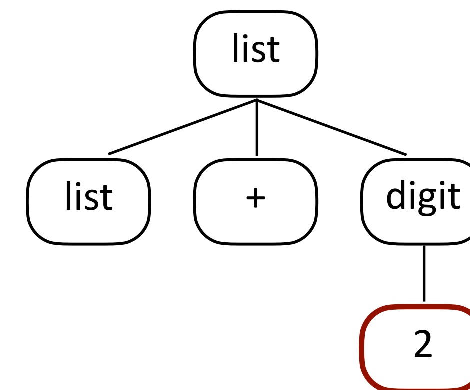
Context-Free Grammars

Right-most Derivation

```
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 + X**

```
S. <list>
1. <list>            + <digit>
4. <list>            +   2
```

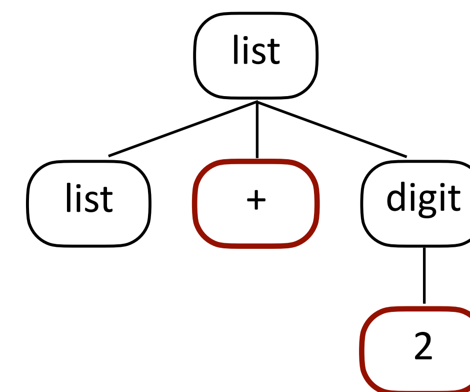


Context-Free Grammars

Right-most Derivation

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

```
S. <list>
1. <list>          + <digit>
4. <list>          +  2
```



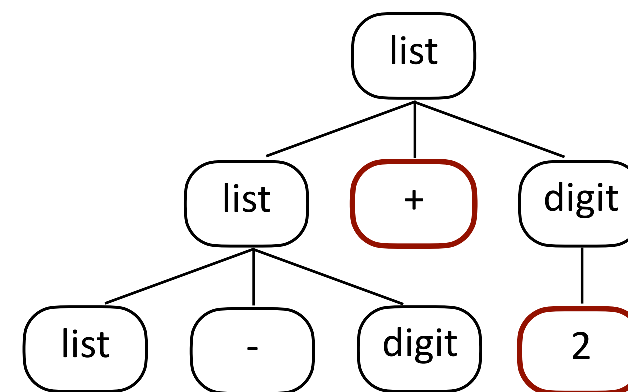
Context-Free Grammars

Right-most Derivation

```
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>
4. <list>              + 2
2. <list>  - <digit>   + 2
```



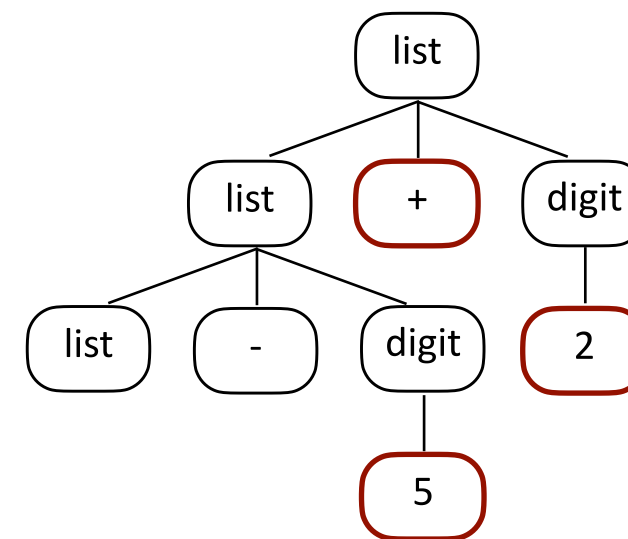
Context-Free Grammars

Right-most Derivation

```
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 - X X X

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



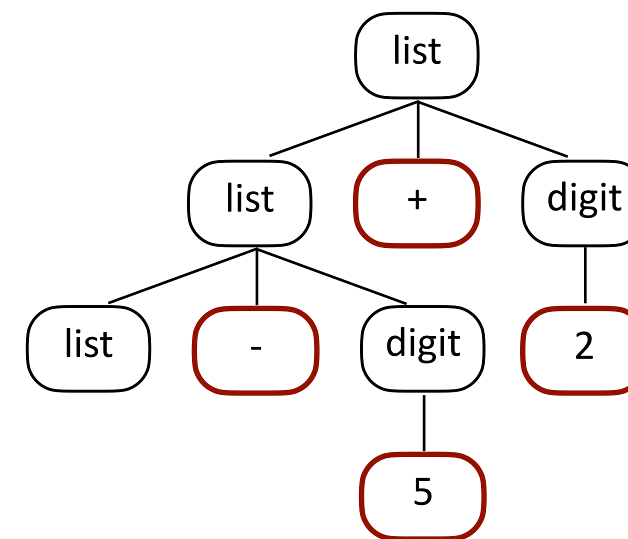
Context-Free Grammars

Right-most Derivation

```
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

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



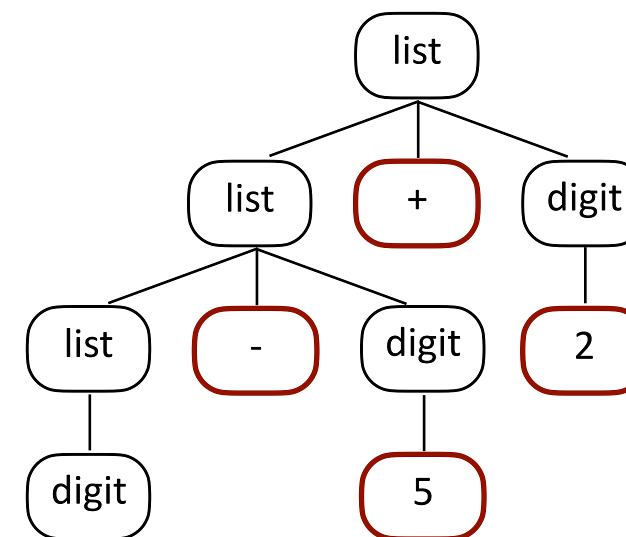
Context-Free Grammars

Right-most Derivation

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



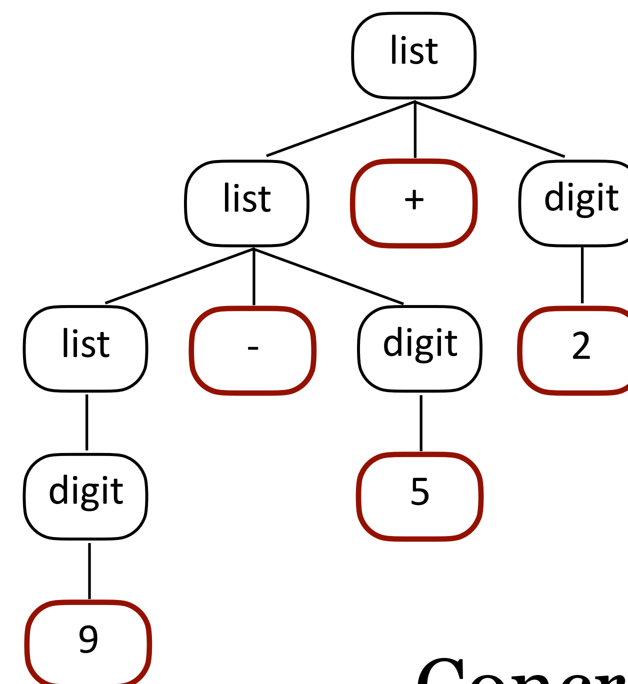
Context-Free Grammars

Right-most Derivation

```
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: X X X X X

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



Concrete Syntax Tree (CST)

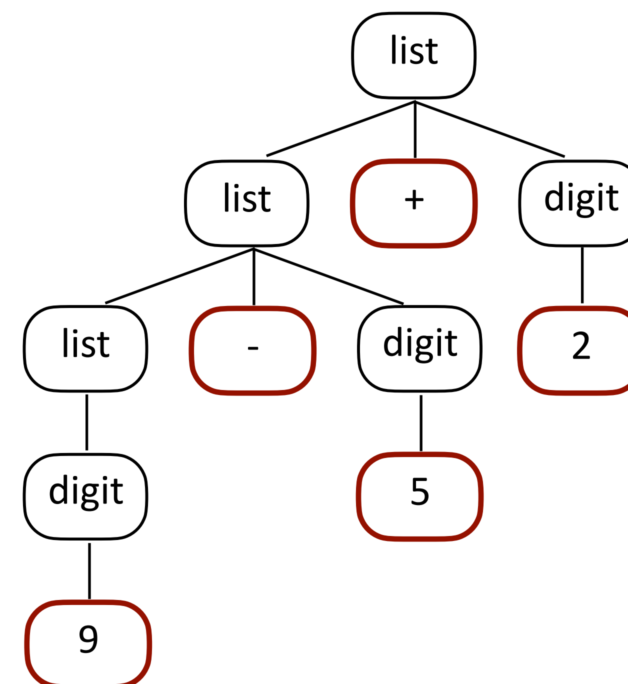
Context-Free Grammars

Right-most Derivation

```
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: X X X X X

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

Context-Free Grammars

Derivations

```
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:

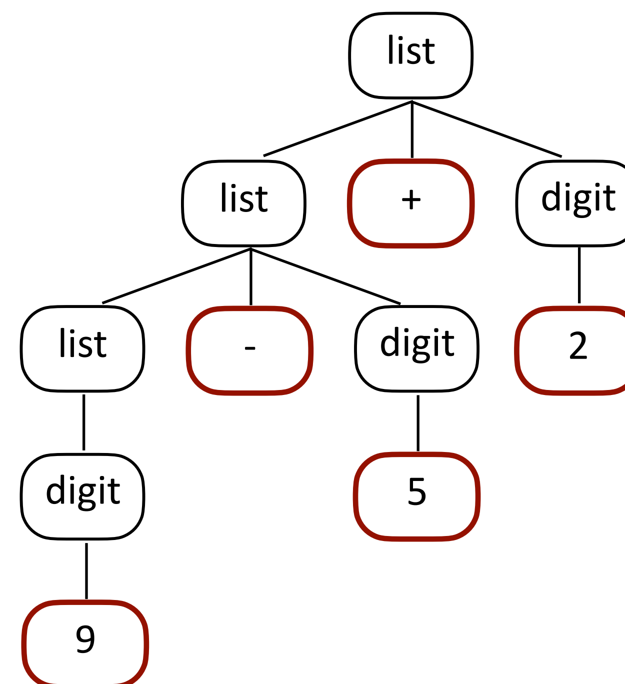
Earlier slide

```
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:

```
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.



Note the CST resulting from the left-most derivation is the same as the CST resulting from the right-most derivation. It's nice when that works out, as the grammar is unambiguous.

Tiny English Grammar

```
1.  <sentence>    → <subject> <verb-phrase> <object>.
2.  <subject>     → This
3.                → Computers
4.                → I
5.  <verb-phrase> → <adverb> <verb>
6.                → <verb>
7.  <adverb>      → never
8.  <verb>        → is | run | am | tell
9.  <object>      → the <noun>
10.               → a <noun>
11.               → <noun>
12. <noun>        → grammar | platypus | cheese | walrus
```

Tiny English Grammar

```
1. <sentence>    → <subject> <verb-phrase> <object>.
2. <subject>     → This
3.               → Computers
4.               → I
5. <verb-phrase> → <adverb> <verb>
6.               → <verb>
7. <adverb>      → never
8. <verb>        → is | run | am | tell
9. <object>      → the <noun>
10.              → a <noun>
11.              → <noun>
12.<noun>        → grammar | platypus | cheese | walrus
```

I am the walrus.

Legal? Prove it.

Tiny English Grammar

```
1.  <sentence>     → <subject> <verb-phrase> <object>.
2.  <subject>      → This
3.                 → Computers
4.                 → I
5.  <verb-phrase>  → <adverb> <verb>
6.                 → <verb>
7.  <adverb>       → never
8.  <verb>         → is | run | am | tell
9.  <object>       → the <noun>
10.                → a <noun>
11.                → <noun>
12. <noun>         → grammar | platypus | cheese | walrus
```

This is a grammar.

Legal? Prove it.

Tiny English Grammar

```
1.  <sentence>    → <subject> <verb-phrase> <object>.
2.  <subject>     → This
3.                → Computers
4.                → I
5.  <verb-phrase> → <adverb> <verb>
6.                → <verb>
7.  <adverb>      → never
8.  <verb>        → is | run | am | tell
9.  <object>      → the <noun>
10.               → a <noun>
11.               → <noun>
12. <noun>        → grammar | platypus | cheese | walrus
```

Computers never run the platypus?

Legal? Prove it.

Tiny English Grammar

```
1.  <sentence>    → <subject> <verb-phrase> <object>.
2.  <subject>     → This
3.                → Computers
4.                → I
5.  <verb-phrase> → <adverb> <verb>
6.                → <verb>
7.  <adverb>      → never
8.  <verb>        → is | run | am | tell
9.  <object>      → the <noun>
10.               → a <noun>
11.               → <noun>
12. <noun>        → grammar | platypus | cheese | walrus
```

Computers never run the platypus?

Legal? No. The "?" is not part of the grammar. This is a lex error.

If "?" were in Σ then this would be a parse error. What error message would the parser emit?

Tiny English Grammar

```
1.  <sentence>     → <subject> <verb-phrase> <object>.
2.  <subject>      → This
3.                 → Computers
4.                 → I
5.  <verb-phrase>  → <adverb> <verb>
6.                 → <verb>
7.  <adverb>       → never
8.  <verb>         → is | run | am | tell
9.  <object>       → the <noun>
10.                → a <noun>
11.                → <noun>
12. <noun>         → grammar | platypus | cheese | walrus
```

I hate cheese.

Legal? Prove it.

Tiny English Grammar

```
1. <sentence>    → <subject> <verb-phrase> <object>.
2. <subject>     → This
3.               → Computers
4.               → I
5. <verb-phrase> → <adverb> <verb>
6.               → <verb>
7. <adverb>      → never
8. <verb>        → is | run | am | tell
9. <object>      → the <noun>
10.              → a <noun>
11.              → <noun>
12.<noun>        → grammar | platypus | cheese | walrus
```

I hate cheese.

Legal? No. The “hate” terminal is not in our grammar. But if it were, what error message would the parser emit?

Parse error on line 1: Expected {*never*, *is*, *run*, *am*, *tell*} but found *hate*.

Tiny Expression Grammar

```
1. <goal> → <expr>
2. <expr> → <expr> <op> <term>
3.        → <term>
4. <term> → num
5.        → id
6. <op>   → +
7.        → -
```

S = <goal>

N = { <goal>, <expr>, <term>, <op> }

T = { num, id, +, - }

P = { 1, 2, 3, 4, 5, 6, 7 }

Tiny Expression Grammar

```
1. <goal> → <expr>
2. <expr> → <expr> <op> <term>
3.        → <term>
4. <term> → num
5.        → id
6. <op>   → +
7.        → -
```

input tokens: **x + 2 - y**

Left-most derivation

```
<goal>
<expr>
<expr> <op> <term>
<expr> <op> <term> <op> <term>
<term> <op> <term> <op> <term>
[id,x] <op> <term> <op> <term>
[id,x]   +  <term> <op> <term>
[id,x]   +  [num,2]<op> <term>
[id,x]   +  [num,2]  -  <term>
[id,x]   +  [num,2]  -  [id,y]
```

Tiny Expression Grammar

```
1. <goal> → <expr>
2. <expr> → <expr> <op> <term>
3.        → <term>
4. <term> → num
5.        → id
6. <op>   → +
7.        → -
```

input tokens: `x + 2 - y`

Left-most derivation	Right-most derivation
<goal>	<goal>
<expr>	<expr>
<expr> <op> <term>	<expr> <op> <term>
<expr> <op> <term> <op> <term>	<expr> <op> [id,y]
<term> <op> <term> <op> <term>	<expr> - [id,y]
[id,x] <op> <term> <op> <term>	<expr> <op> <term> - [id,y]
[id,x] + <term> <op> <term>	<expr> <op> [num,2] - [id,y]
[id,x] + [num,2]<op> <term>	<expr> + [num,2] - [id,y]
[id,x] + [num,2] - <term>	<term> + [num,2] - [id,y]
[id,x] + [num,2] - [id,y]	[id,x] + [num,2] - [id,y]

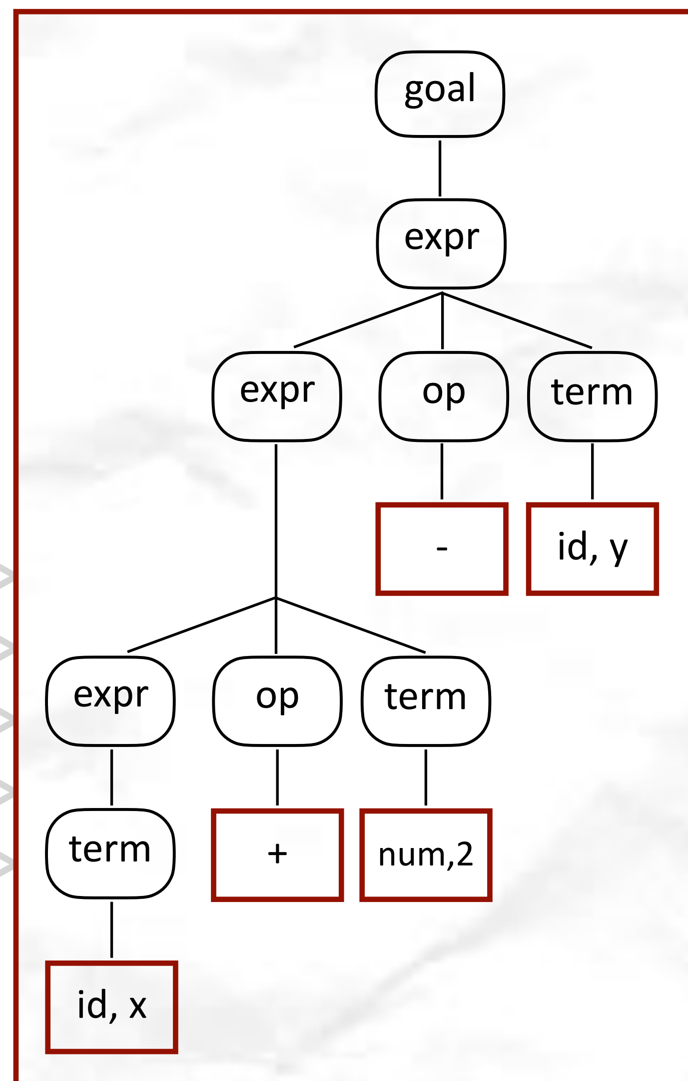
Tiny Expression Grammar

```
1. <goal> → <expr>
2. <expr> → <expr> <op> <term>
3.        → <term>
4. <term> → num
5.        → id
6. <op>   → +
7.        → -
```

input tokens: **x + 2 - y**

Left-most derivation

```
<goal>
<expr>
<expr> <op> <term>
<expr> <op> <term> <op>
<term> <op> <term> <op>
[id,x] <op> <term> <op>
[id,x]   +  <term> <op>
[id,x]   +  [num,2]<op>
[id,x]   +  [num,2]  -
[id,x]   +  [num,2]  -
```



Concrete Syntax Tree

Right-most derivation

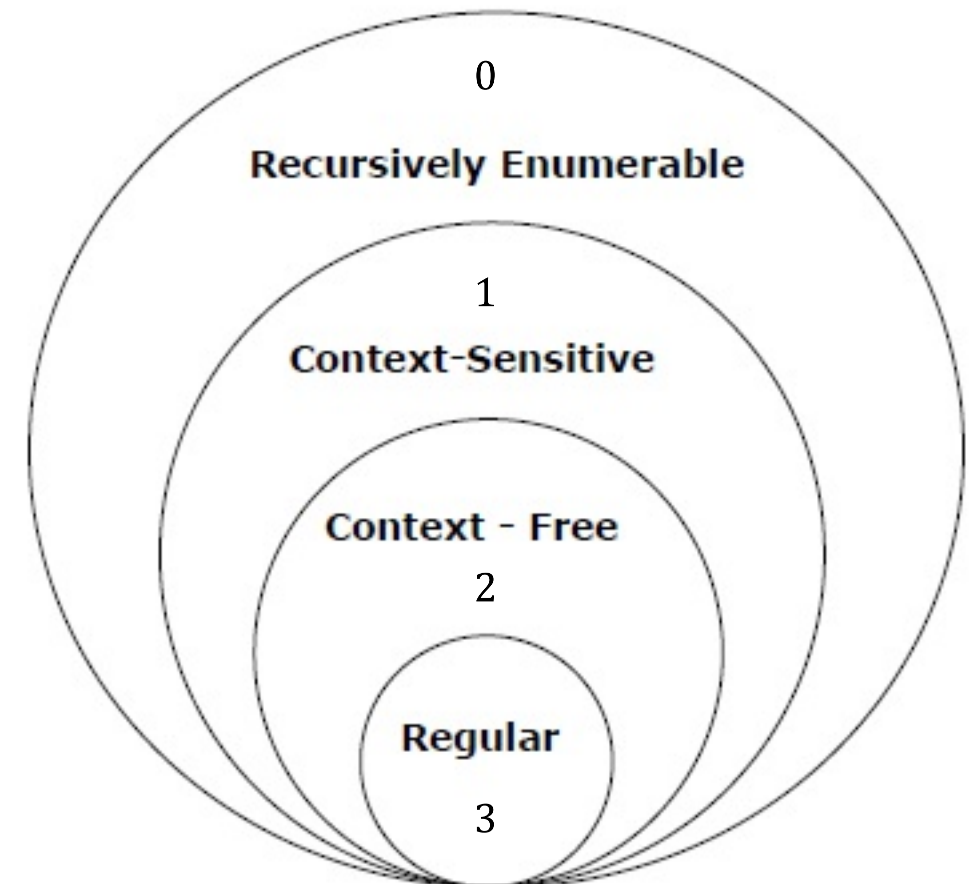
```
                          <goal>
                          <expr>
     <expr> <op>  <term>
     <expr> <op>  [id,y]
     <expr>   -   [id,y]
<op> <term>   -   [id,y]
<op> [num,2]  -   [id,y]
 +   [num,2]  -   [id,y]
 +   [num,2]  -   [id,y]
 +   [num,2]  -   [id,y]
```

Lexing vs. Parsing

Where do we draw the line between lexing (Regular Expressions) and parsing (Context-Free Grammars)?

Regular Expressions are used to **classify** tokens using patterns

- id, num, keywords, etc.
- more concise and simple than a CFG for this purpose
- more efficient than a CFG for this purpose
- cannot count



Context-Free Grammars can count (but how?)

- enforce `{ }`, `( )`, `begin end`, `if then else`, etc.
- detect and enforce structure (for expressions)

Predictive Parsing

Remember how much fun it was to backtrack when we followed a wrong path during a parse?

No?

Neither do I.

It would be nice if we could parse grammars that allow us to predict the correct path and never have to backtrack.

Predictive Parsing

… relies on data about the **first** tokens that can be generated by a production to make decisions about which production to follow for the next input token.

Grammars that permit predictive parsing by reading the tokens
  Left-to-right, while doing a
  Left-most derivation, using only
  1 token look-ahead
are classified as LL(1) grammars.

Grammars that require us to look ahead $k > 1$ tokens for predictive parsing are called LL(k).

Predictive Parsing - LL(1) Grammar

Grammars that permit predictive parsing by reading the tokens
Left-to-right, while doing a
Left-most derivation, considering only
1 token at a time
are classified as LL(1) grammars.

Here's an example:

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

Remember ε?
It's the empty string.
This is an ε-production.

Predictive Parsing - LL(1) Grammar

Here's an LL(1) grammar:

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

Predictive parsing is easy* for LL(1) grammars. We can use a technique called **Recursive Descent**.

*Well, it's easy if you love recursion. But we all do, right?

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

To implement a Recursive Descent parser we need write a routine for each non-terminal in the grammar and a *match()* routine to consume tokens from the input.

```
parseStart()
parseValue()
parseExpr()
parseValues()

match()
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseStart() {
   parseValue()
   match($)
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}

parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValues() {
  if token in {num,(}
      parseValue()
      parseValues()
   else
      // nothing
      // it's a ε
      // production
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValues() {
  if token in {num,(}
    parseValue()
    parseValues()
  else
    // nothing
    // it's a ε
    // production
  end if
}
```

```
parseStart() {
   parseValue()
   match($)
}

parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

The <values> production begins with <value>. The first tokens in a <value> production are **num** and **(**.

In other words, the **first set** of <values> is {**num, (**}.

FIRST(values) = {**num, (**}

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValues() {
  if token in {num,(}
      parseValue()
      parseValues()
  else
      // nothing
      // it's a ε
      // production
  end if
}
```

The ε production is where the recursion stops. Even though there is no code, this is a great place for comments explaining what's going on.

```
parseStart() {
   parseValue()
   match($)
}

parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
match( expectedTokens ) {
   if currentToken in expectedTokens
      consume currentToken
      inc tokenPointer
   else
      error: expected expectedTokens
                but found currentToken.
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

```
parseValues() {
   if token in {num,(}
      parseValue()
      parseValues()
   else
      // nothing
      // it’s a ε
      // production
   end if
}
```

Recursive Descent Parsing Our Grammar

Our Language Grammar

```
Program              ::== Block $
Block                ::== { StatementList }
StatementList        ::== Statement StatementList
                     ::== ε
Statement            ::== PrintStatement
                     ::== AssignmentStatement
                     ::== VarDecl
                     ::== WhileStatement
                     ::== IfStatement
                     ::== Block
PrintStatement       ::== print ( Expr )
AssignmentStatement  ::== Id = Expr
VarDecl              ::== type Id
WhileStatement       ::== while BooleanExpr Block
IfStatement          ::== if BooleanExpr Block

Expr                 ::== IntExpr
                     ::== StringExpr
                     ::== BooleanExpr
                     ::== Id
IntExpr              ::== digit intop Expr
                     ::== digit
StringExpr           ::== " CharList "
BooleanExpr          ::== ( Expr boolop Expr )
                     ::== boolval
Id                   ::== char
CharList             ::== char CharList
                     ::== space CharList
                     ::== ε

type                 ::== int | string | boolean
char                 ::== a | b | c ... z
space                ::== the space character
digit                ::== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

boolop               ::== == | !=
boolval              ::== false | true
intop                ::== +

Comments are bounded by /* and */ and ignored by the lexer.
```

Curly braces denote scope.

= is assignment.

== is test for equality.

```
parseProgram() {
   parseBlock()
   match($)
}

parseBlock() {
   match({)
   parseStatementList()
   match(})
}
   ⋮
parsePrintStatement() {
   match(print)
   match( ( )
   parseExpr()
   match( ) )
}
   ⋮
```

This version matches the symbols in the grammar to illustrate the parse code. But it's better if you use your token names.

Recursive Descent Parsing Our Grammar

Our Language Grammar

```
Program              ::== Block $
Block                ::== { StatementList  }
StatementList        ::== Statement StatementList
                     ::== ε
Statement            ::== PrintStatement
                     ::== AssignmentStatement
                     ::== VarDecl
                     ::== WhileStatement
                     ::== IfStatement
                     ::== Block
PrintStatement       ::== print ( Expr )
AssignmentStatement  ::== Id = Expr
VarDecl              ::== type Id
WhileStatement       ::== while BooleanExpr Block
IfStatement          ::== if BooleanExpr Block

Expr                 ::== IntExpr
                     ::== StringExpr
                     ::== BooleanExpr
                     ::== Id
IntExpr              ::== digit intop Expr
                     ::== digit
StringExpr           ::== " CharList "
BooleanExpr          ::== ( Expr boolop Expr )
                     ::== boolval
Id                   ::== char
CharList             ::== char CharList
                     ::== space CharList
                     ::== ε

type                 ::== int | string | boolean
char                 ::== a | b | c ... z
space                ::== the space character
digit                ::== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

boolop               ::== == | !=
boolval              ::== false | true
intop                ::== +

Comments are bounded by /* and */ and ignored by the lexer.
```

Curly braces denote scope.

= is assignment.

== is test for equality.

```
parseProgram() {
   parseBlock()
   match(EOP)
}

parseBlock() {
   match(OPEN_BRACE)
   parseStatementList()
   match(CLOSE_BRACE)
}
   ⋮
parsePrintStatement() {
   match(KEYWORD_PRINT)
   match(OPEN_PAREN)
   parseExpr()
   match(CLOSE_PAREN)
}
   ⋮
```

This version matches the tokens.

Ambiguous Context-Free Grammars

Remember this unambiguous 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:

```
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
```

list, list, +, digit, list, -, digit, 2, digit, 5, 9

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:

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

list, list, +, digit, list, -, digit, 2, digit, 5, 9

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

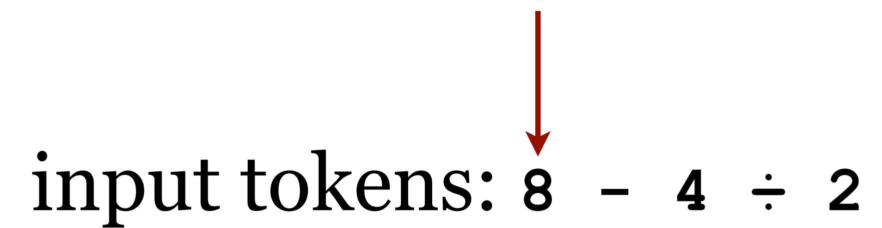
A less cooperative grammar

```
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.        ::== -
```



Left-most derivation

```
<expr>
```

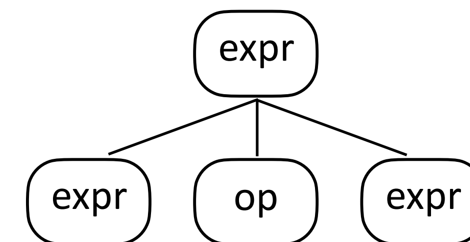
Ambiguous Context-Free Grammars

A less cooperative grammar

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

Left-most derivation

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



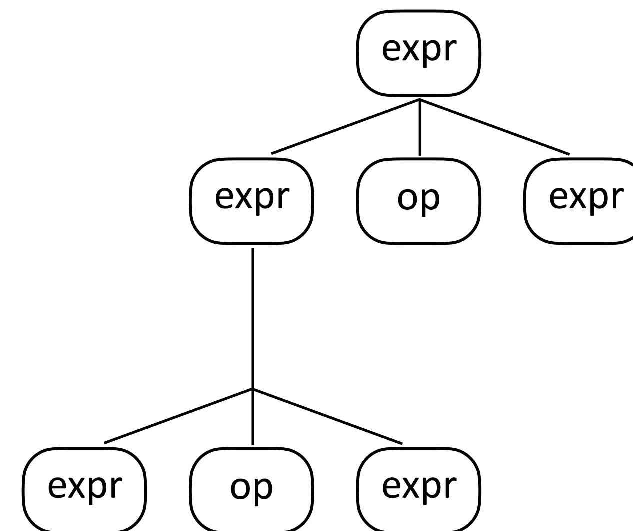
Ambiguous Context-Free Grammars

A less cooperative grammar

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

Left-most derivation

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



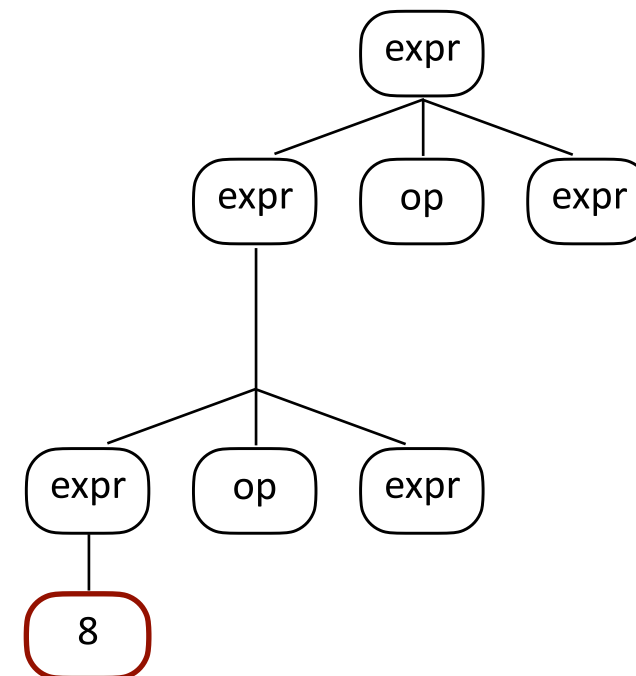
Ambiguous Context-Free Grammars

A less cooperative grammar

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

Left-most derivation

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



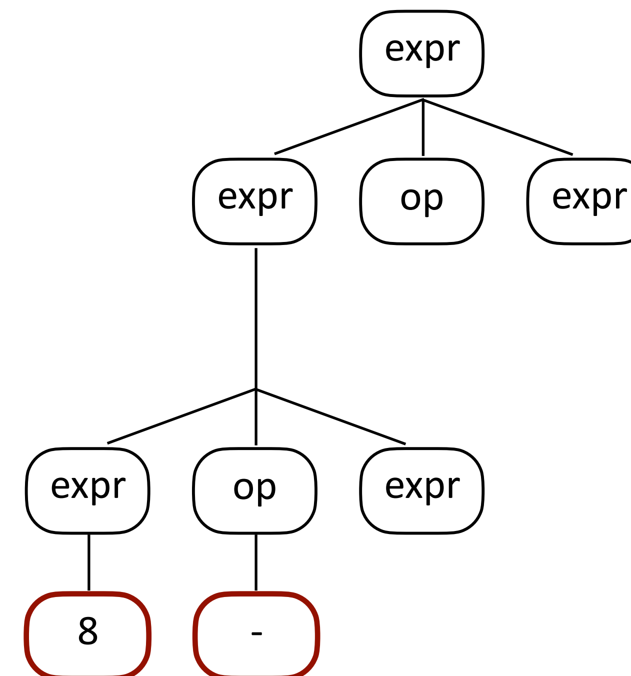
Ambiguous Context-Free Grammars

A less cooperative grammar

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

Left-most derivation

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



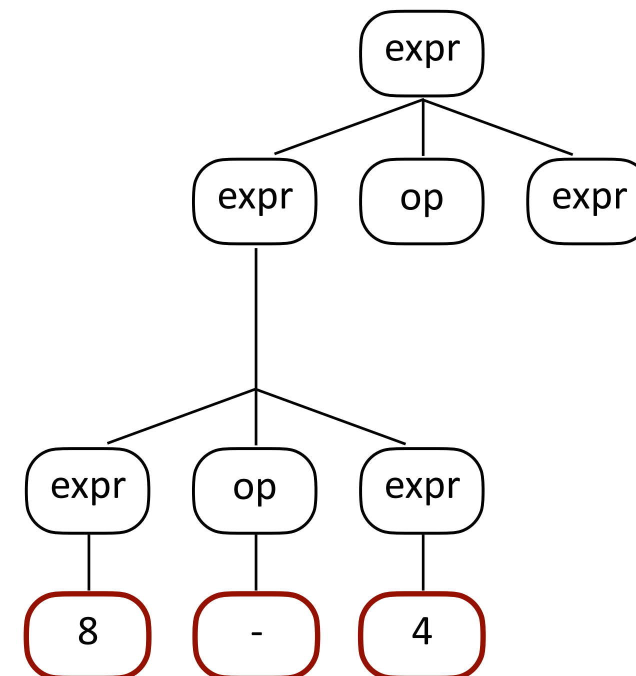
Ambiguous Context-Free Grammars

A less cooperative grammar

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

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

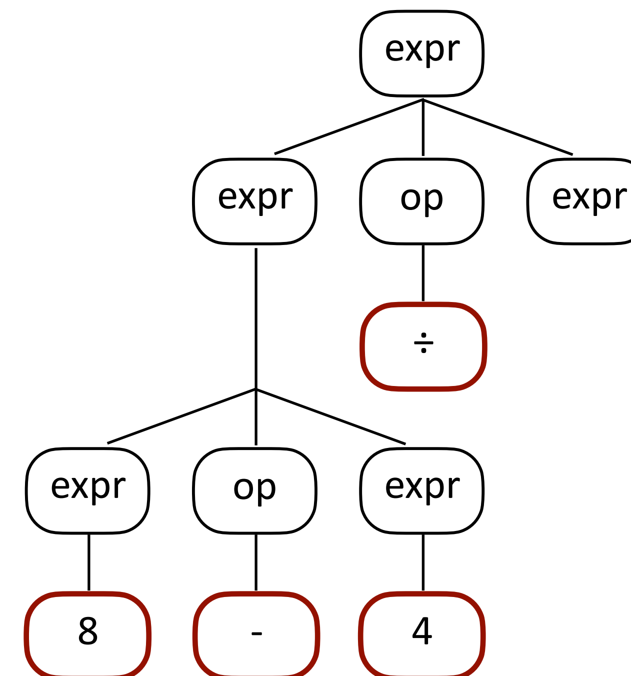
A less cooperative grammar

```
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

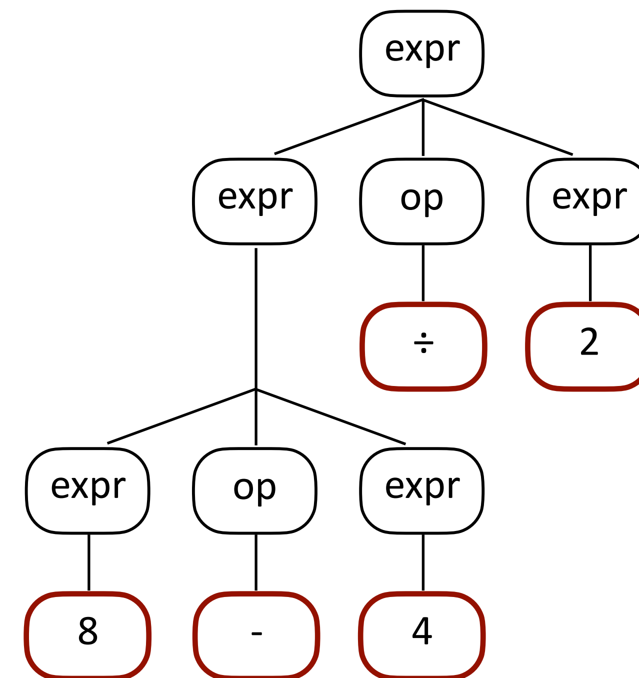
A less cooperative grammar

```
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

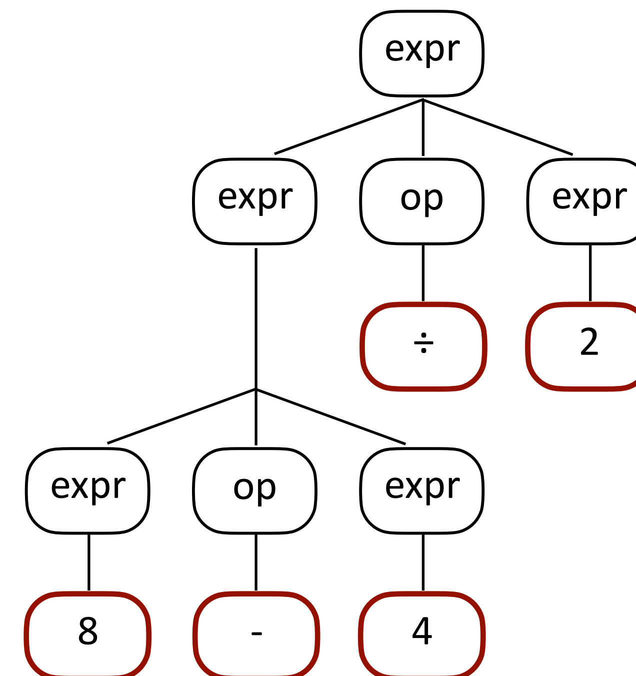
A less cooperative grammar

```
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

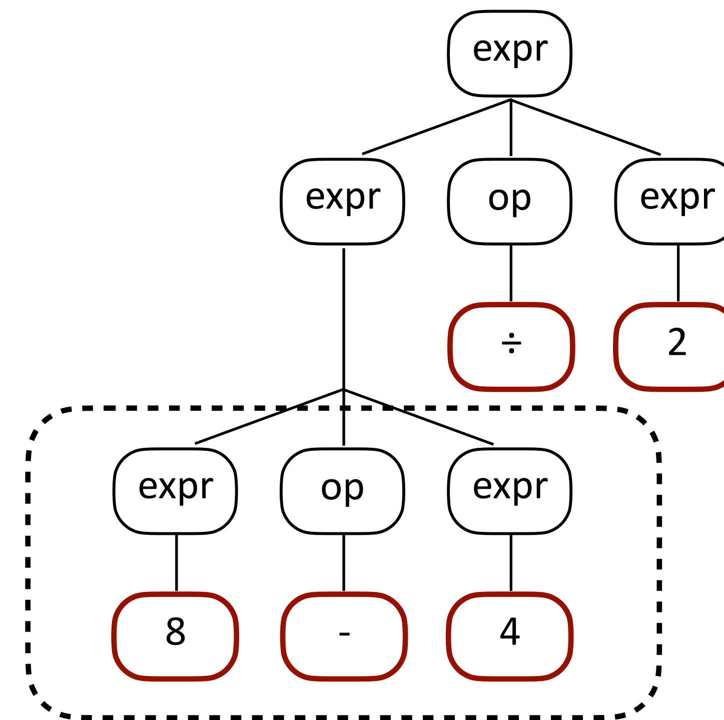
A less cooperative grammar

```
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

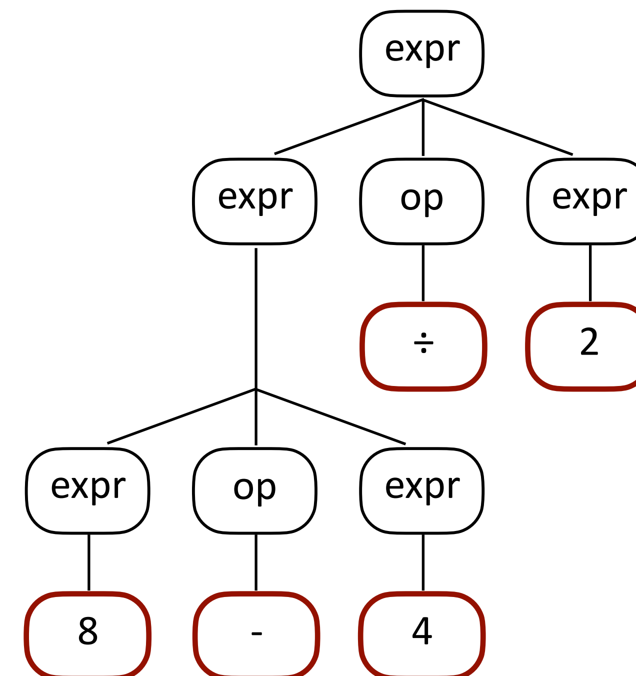
A less cooperative grammar

```
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>

expr

Ambiguous Context-Free Grammars

A less cooperative grammar

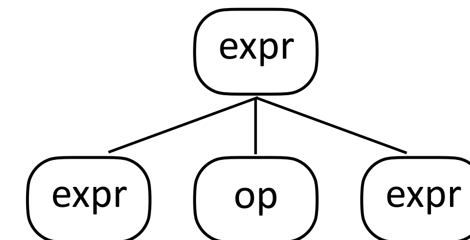
```
1. <expr> ::== <expr> <op> <expr>
2.        ::== num
3. <op>   ::== ÷
4.        ::== -
```

input tokens: **8 - 4 ÷ 2**

Right-most derivation

<expr>

<expr> <op> <expr>



Ambiguous Context-Free Grammars

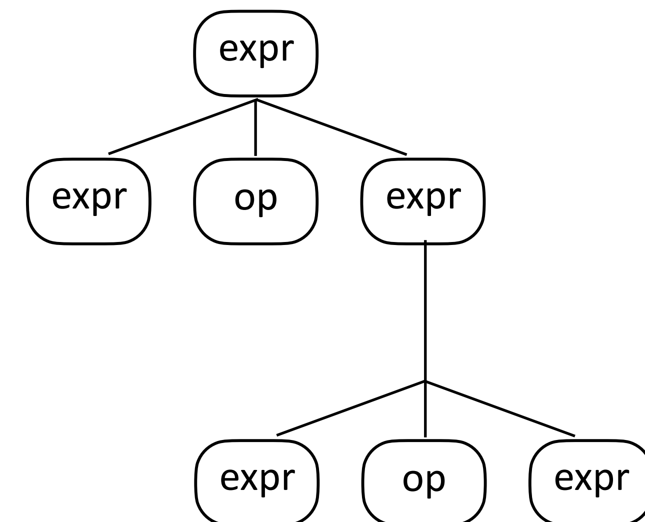
A less cooperative grammar

```
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

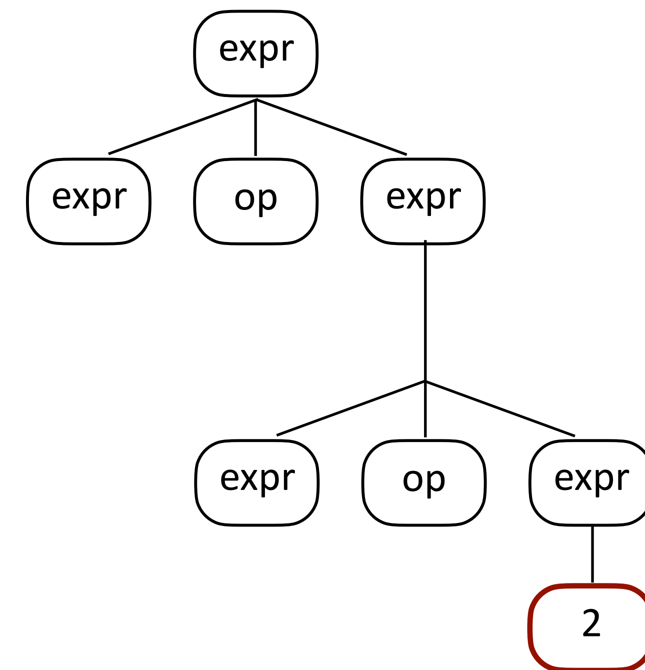
A less cooperative grammar

```
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

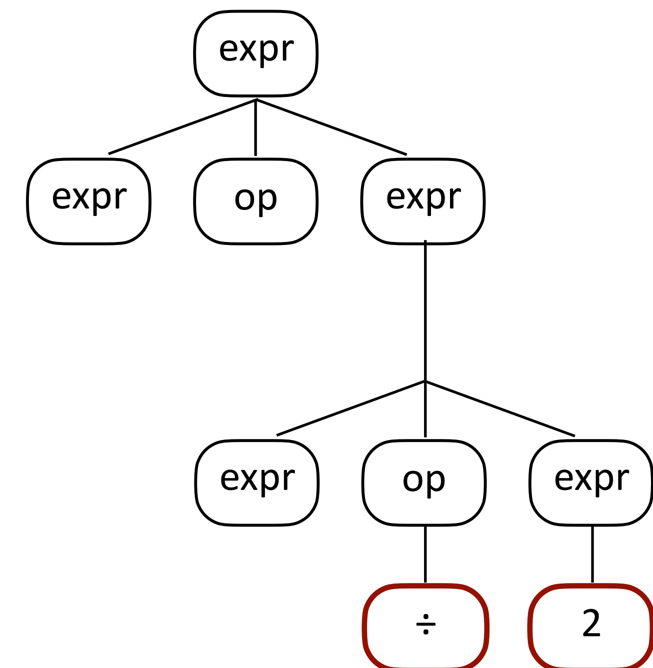
A less cooperative grammar

```
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]
```



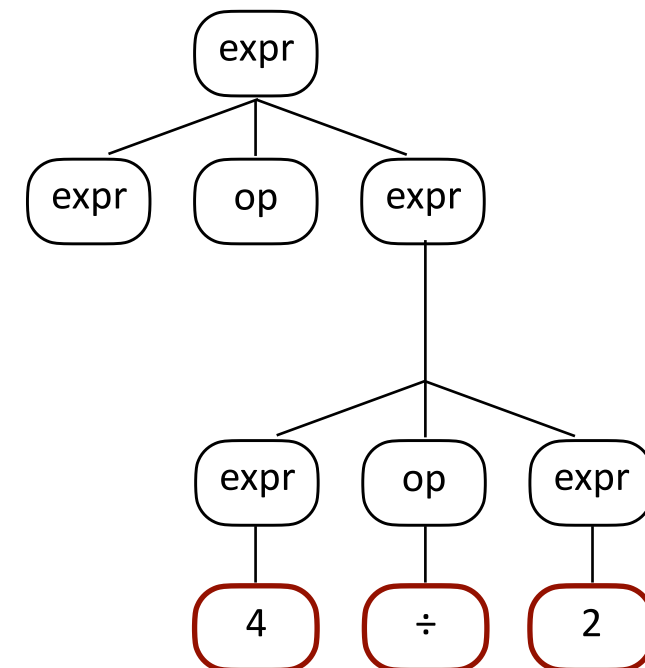
Ambiguous Context-Free Grammars

A less cooperative grammar

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

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

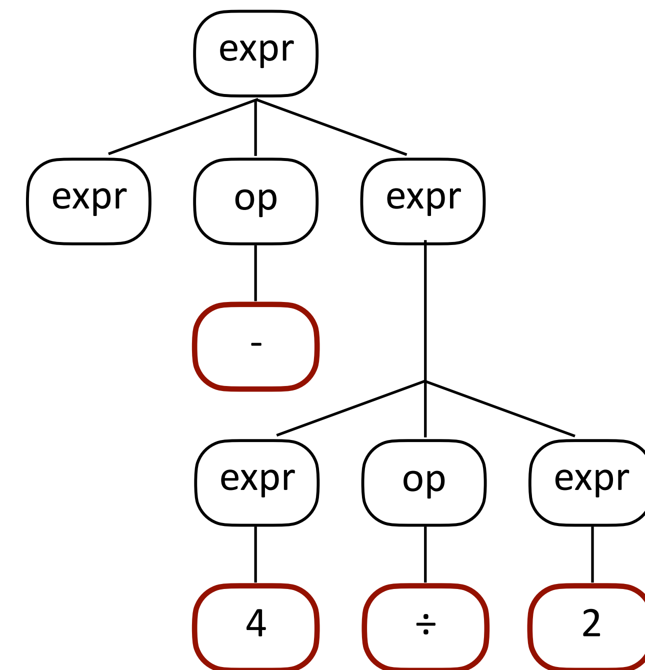
A less cooperative grammar

```
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]
```



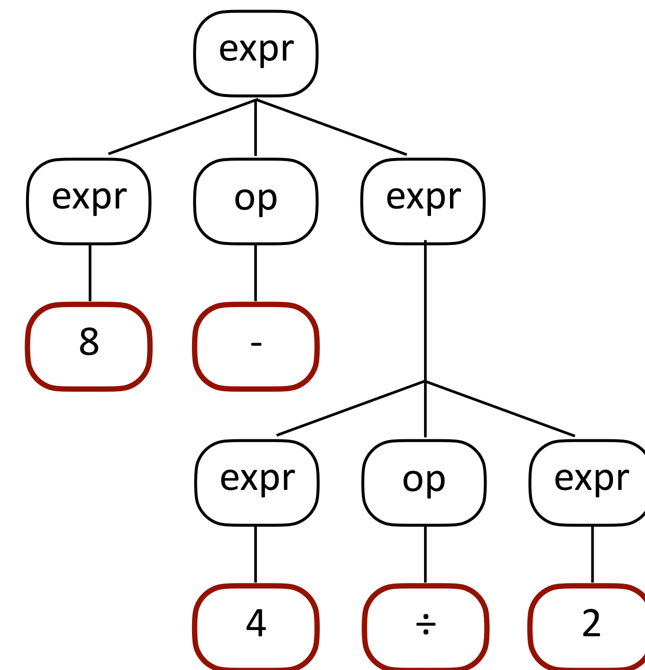
Ambiguous Context-Free Grammars

A less cooperative grammar

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

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

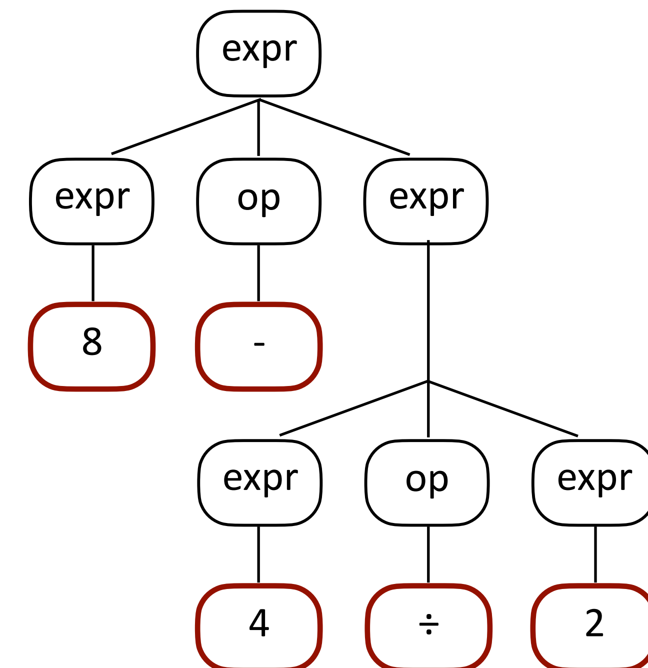
A less cooperative grammar

```
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

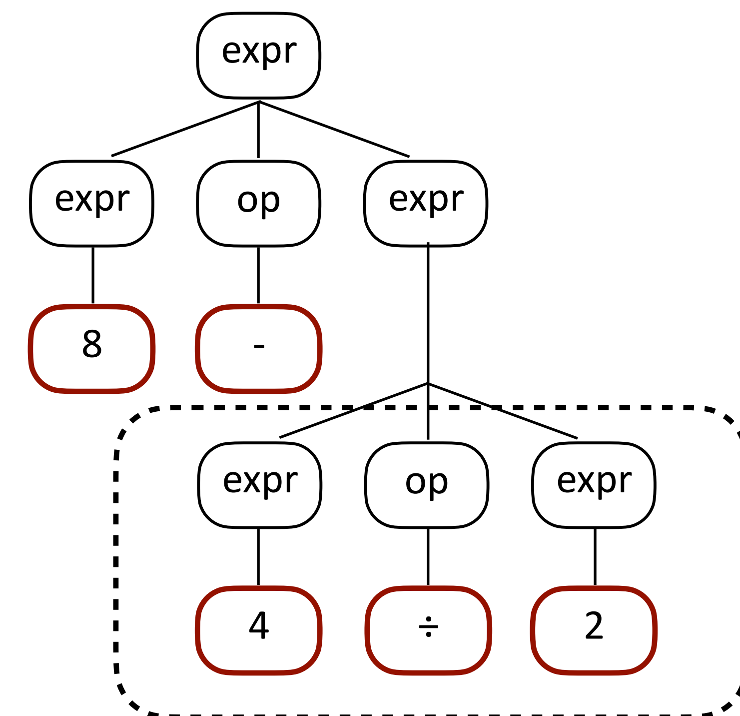
A less cooperative grammar

```
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

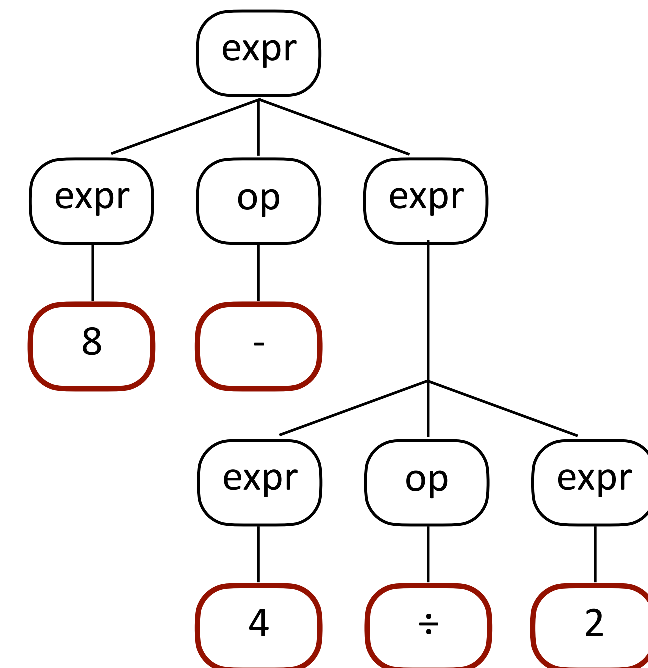
A less cooperative grammar

```
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!

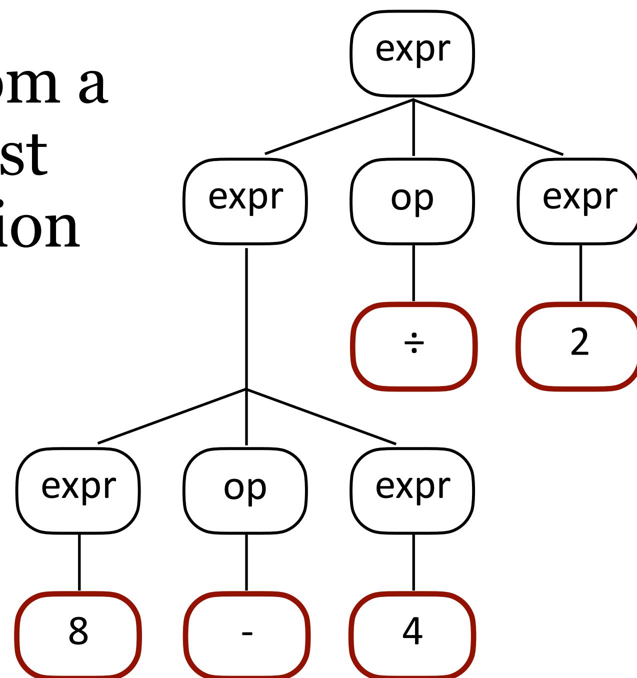
Ambiguous Context-Free Grammars

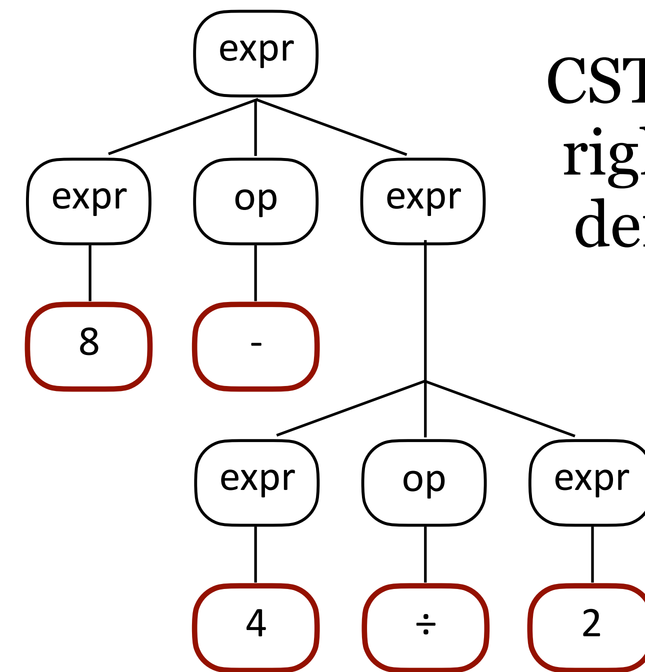
A less cooperative grammar

```
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

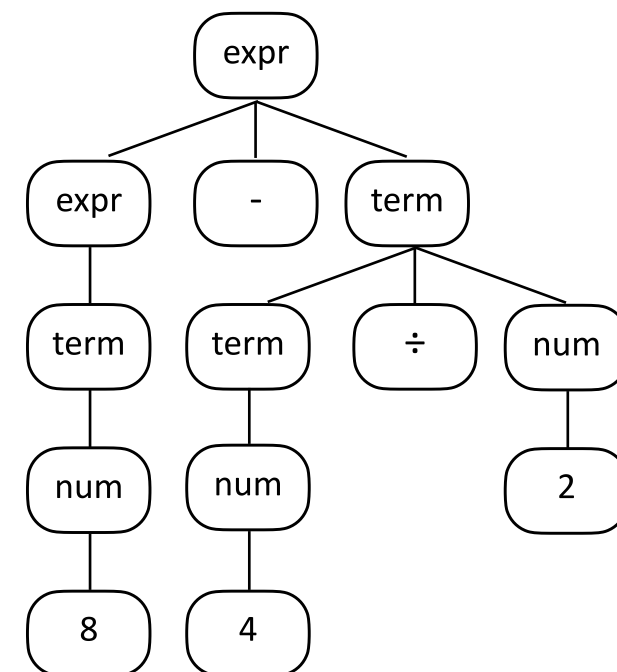
A more cooperative grammar

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



Context-Free Grammars

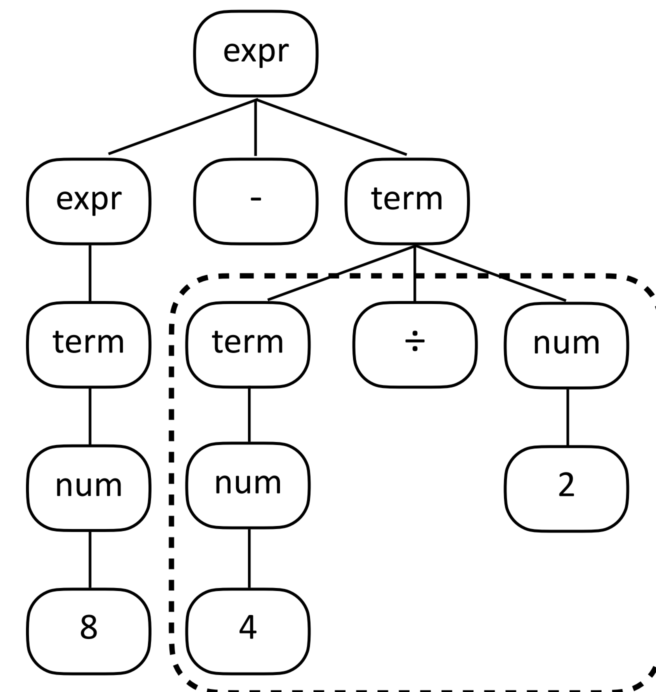
A more cooperative grammar

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



output: **`8 - (4 ÷ 2)`**

Context-Free Grammars

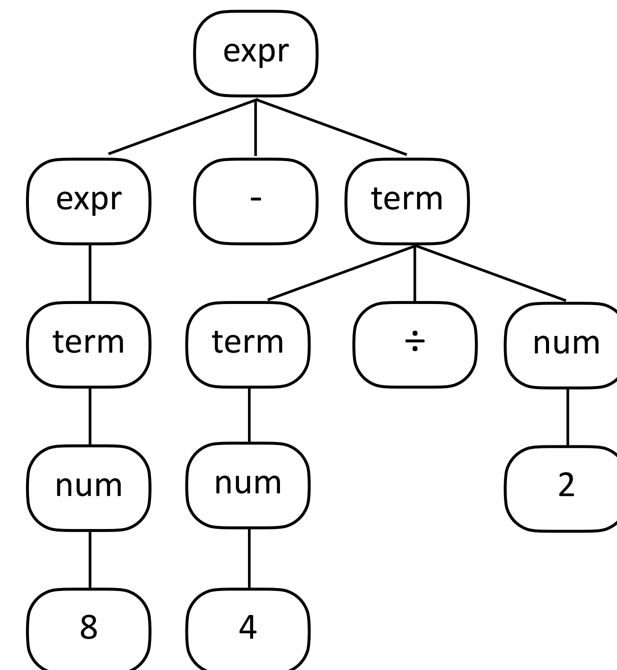
A more cooperative grammar

```
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

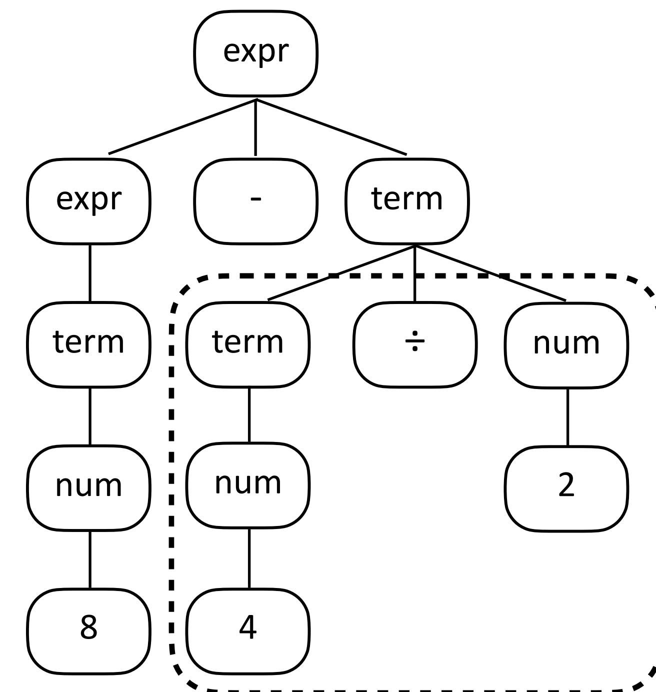
A more cooperative grammar

```
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

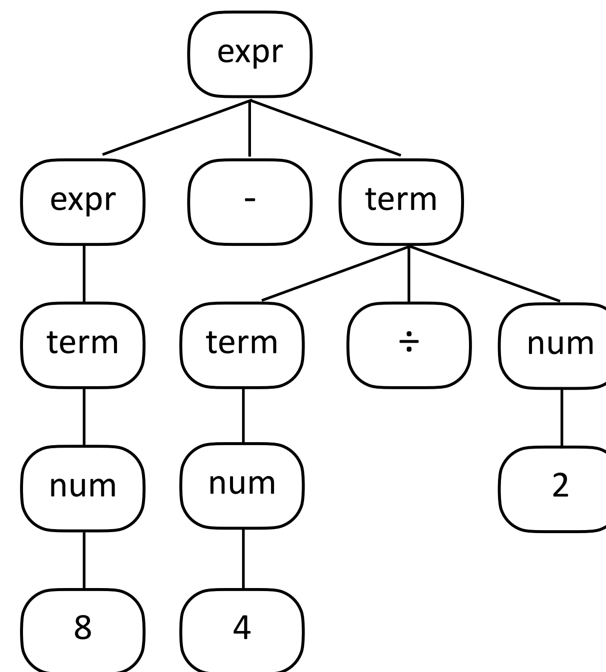
A very cooperative grammar

```
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    ÷ <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
```

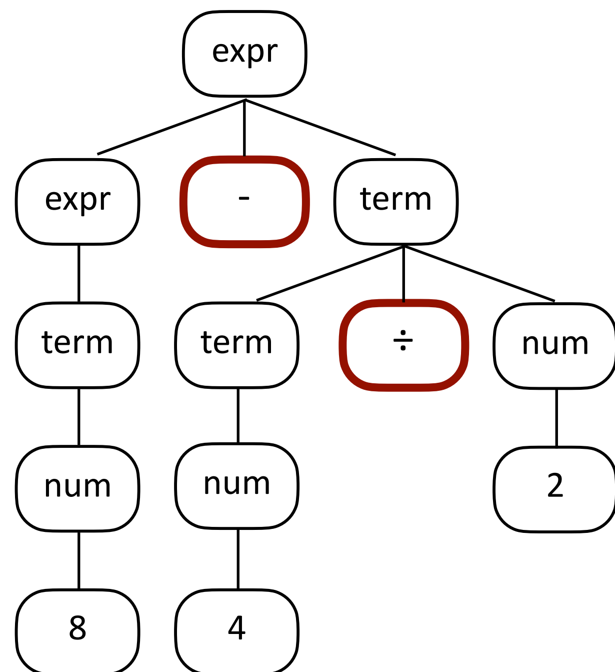
Same CST because
the grammar is unambiguous.

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.

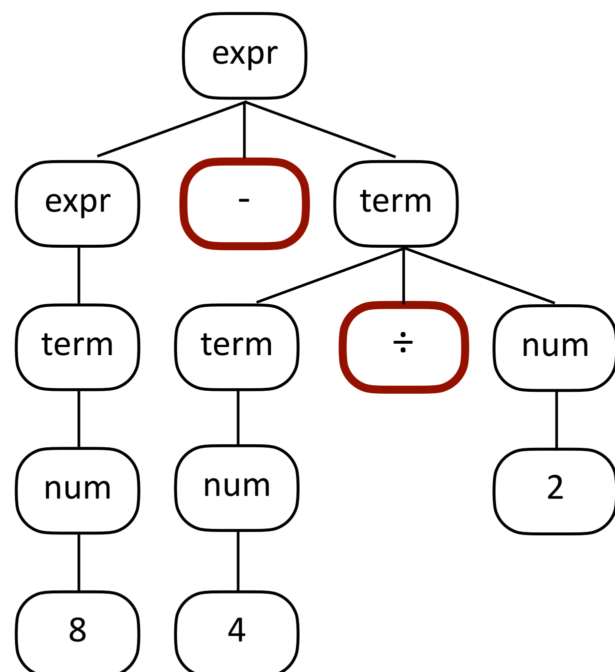
Why is this?

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.

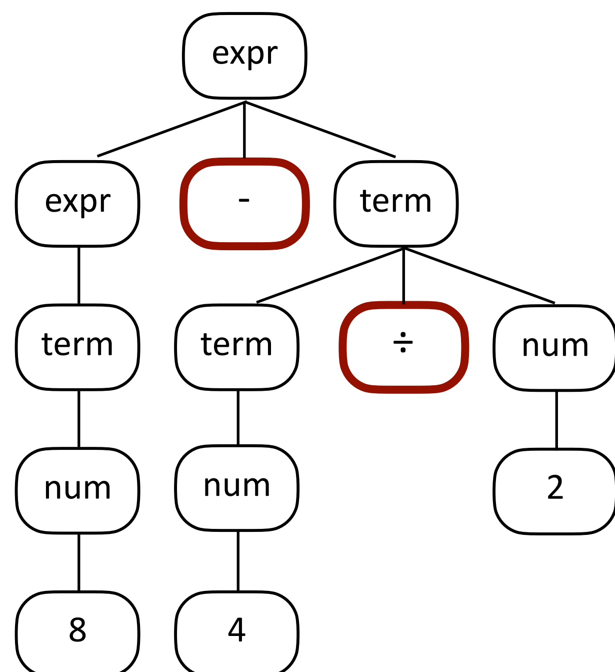
In this example, if there is subtraction, it has to be processed before division because we can only divide **`<term>`**s so we'll have to go through production #1 to get there, thus placing minus higher in the derivation — and therefore higher in the syntax tree — than division.

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.

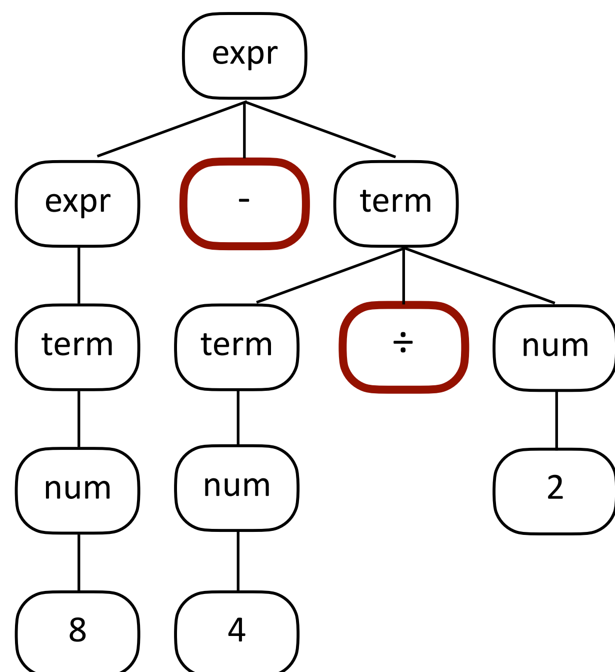
Why do we want this?

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.

Implementing a Concrete Syntax Tree

This is all nice in theory. But how might we build it in practice?

Let's look at another example grammar:

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

Implementing a Concrete Syntax Tree

Aside: Is the grammar ambiguous?

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

Implementing a Concrete Syntax Tree

Aside: Is the grammar LL(1)?

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

Remember . . .
Grammars that permit predictive parsing by reading the tokens
Left-to-right, while doing a
Left-most derivation, considering only
1 token at a time
are classified as LL(1) grammars.

Implementing a Concrete Syntax Tree

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

input tokens: **a + 2**

Left-most derivation

```
<goal>
<expr>   <op> <term>
<term>   <op> <term>
<factor> <op> <term>
[id,a]   <op> <term>
[id,a]    +   <term>
[id,a]    +   <factor>
[id,a]    +   [num,2]
```

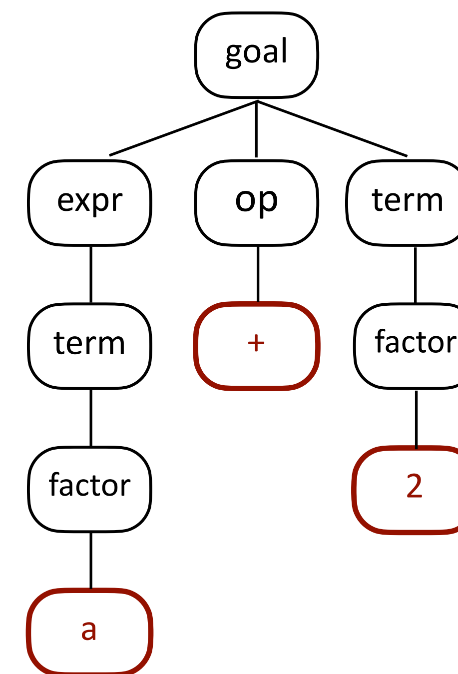
Implementing a Concrete Syntax Tree

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

input tokens: `a + 2`

Left-most derivation

```
<goal>
<expr>   <op> <term>
<term>   <op> <term>
<factor> <op> <term>
[id,a]   <op> <term>
[id,a]    +   <term>
[id,a]    +   <factor>
[id,a]    +   [num,2]
```



CST

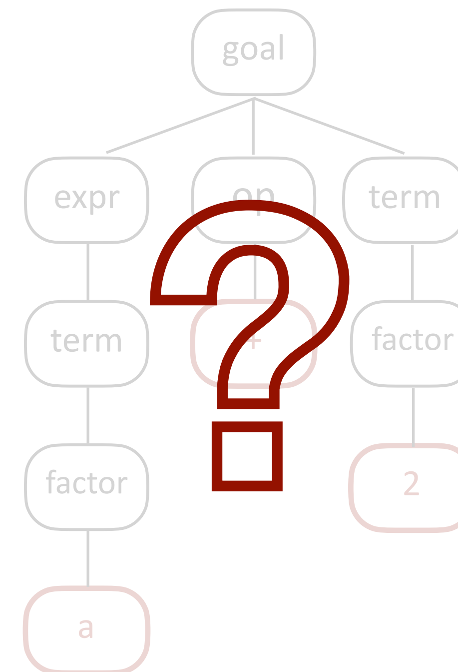
Implementing a Concrete Syntax Tree

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

input tokens: `a + 2`

Left-most derivation

```
<goal>
<expr>   <op> <term>
<term>   <op> <term>
<factor> <op> <term>
[id,a]   <op> <term>
[id,a]    +   <term>
[id,a]    +   <factor>
[id,a]    +   [num,2]
```



Let's build this.

Implementing a Concrete Syntax Tree

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

```
parseGoal(){

  parseExpr()
  parseOp()
  parseTerm()

}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()

}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

```
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

```
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
```

```
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

```
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
```

```
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
```

```
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

```
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

```
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
```

```
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
```

```
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
```

Implementing a Concrete Syntax Tree

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

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
```

Now we need a tree.

```
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

```
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
```

```
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
```

```
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
```

```
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
```

What's a CST node?

```
  }
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
```

CST node

```
name    :
parent  :
children: □□□□…
```

```
  }
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

← Let's not forget about this.

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **`a + 2`**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

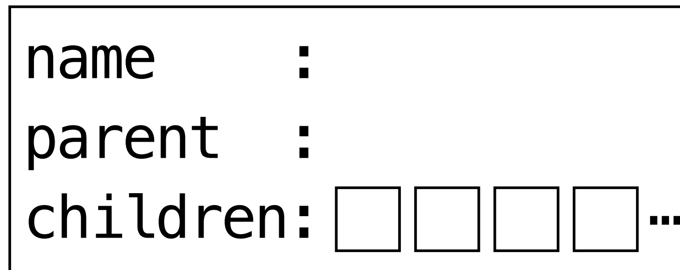
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root = null
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   :
children: □□□□…
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

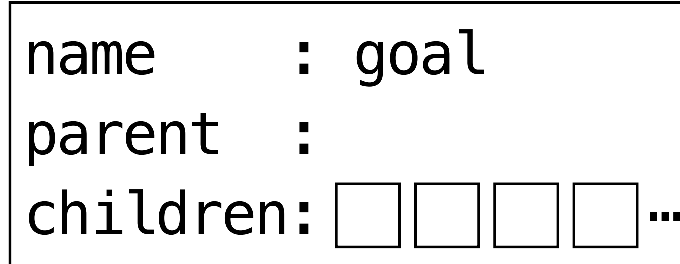
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current = null

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]...
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

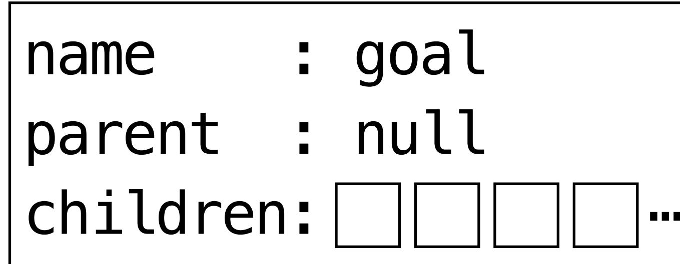
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]…
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

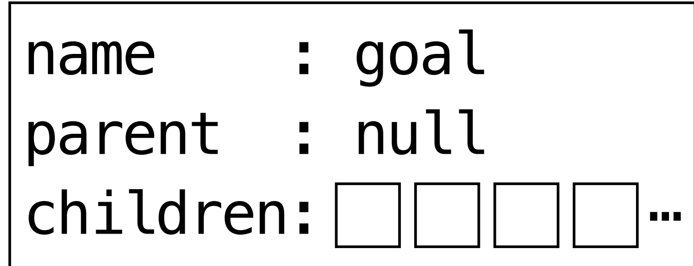
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]…
```

```
name     :
parent   :
children: [ ][ ][ ][ ]…
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

name	: goal
parent	: null
children:	☐☐☐☐…

name	: expr
parent	:
children:	☐☐☐☐…

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]...
```

```
name     : expr
parent   :
children: [ ][ ][ ][ ]...
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

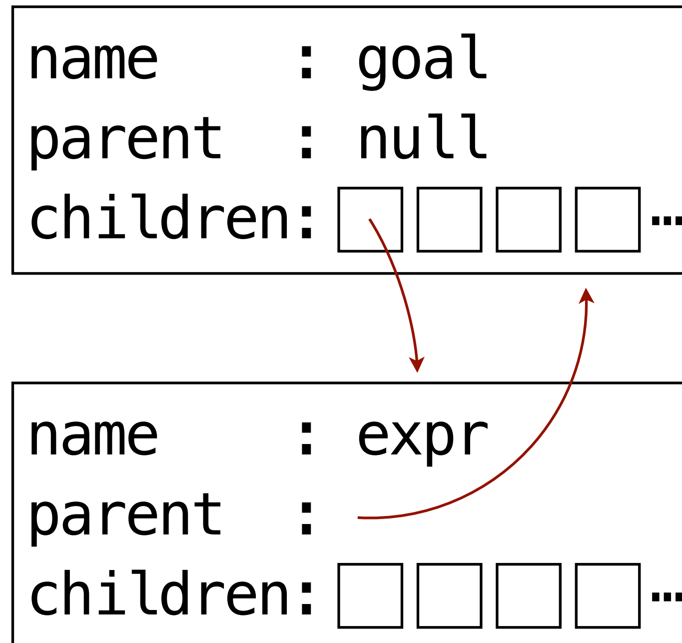
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

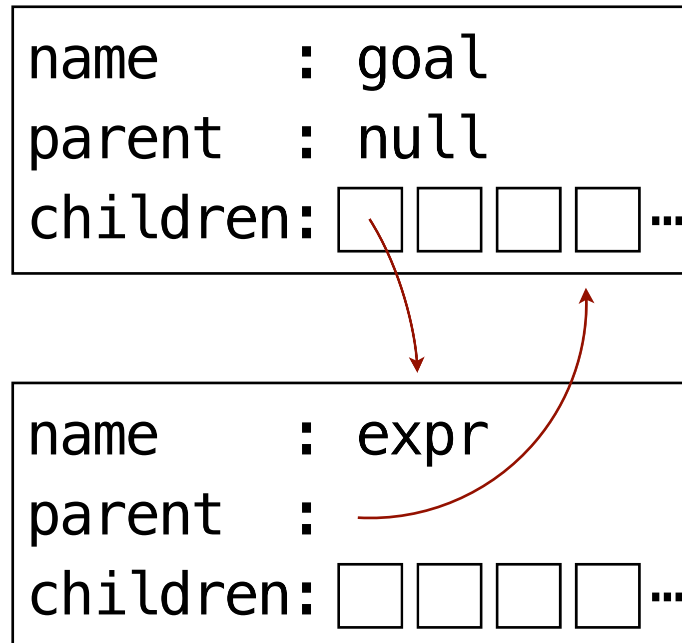
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

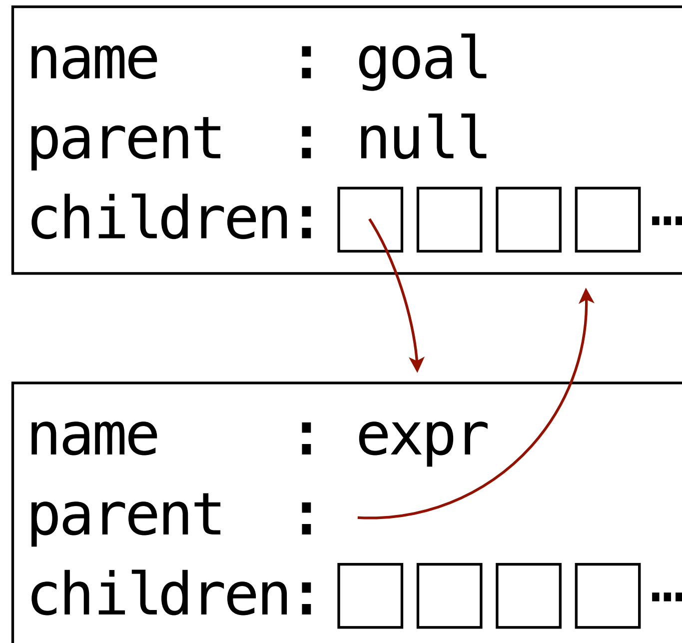
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

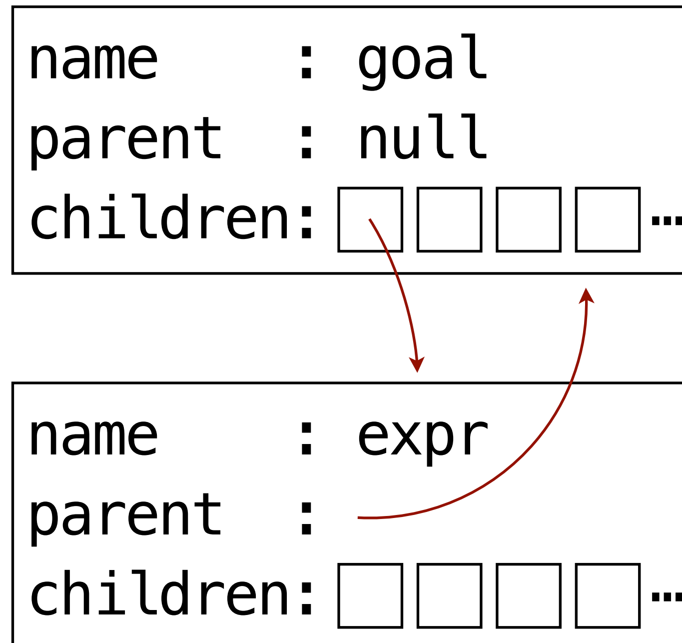
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]…
```

```
name     : expr
parent   :
children: [ ][ ][ ][ ]…
```

```
name     :
parent   :
children: [ ][ ][ ][ ]…
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

```
name     : goal
parent   : null
children: [ ][ ][ ][ ]...

name     : expr
parent   :
children: [ ][ ][ ][ ]...

name     : term
parent   :
children: [ ][ ][ ][ ]...
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

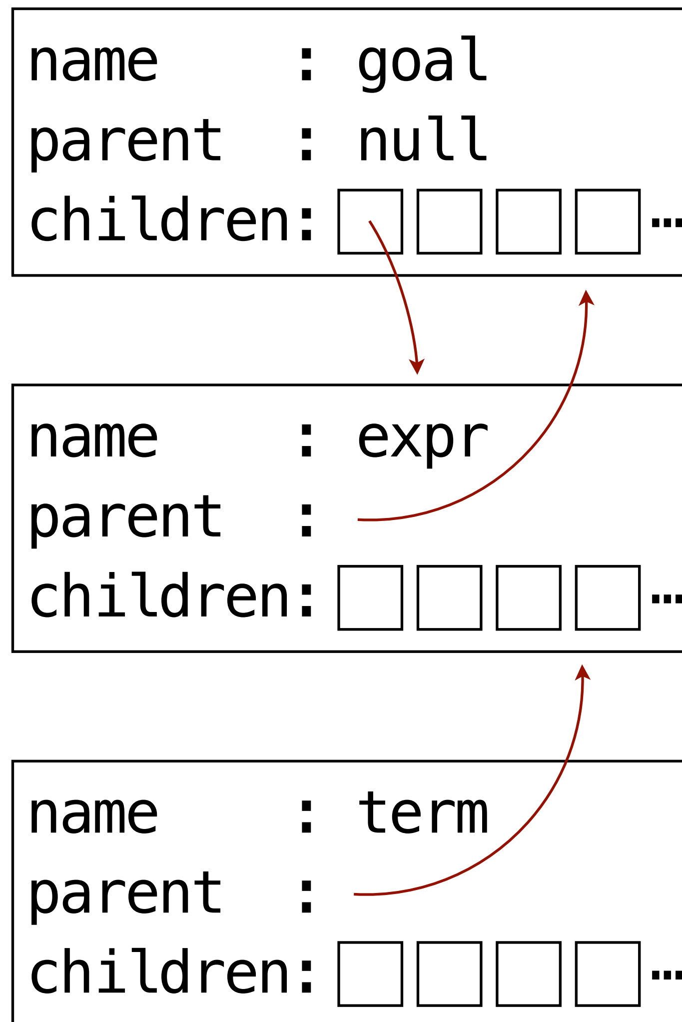
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

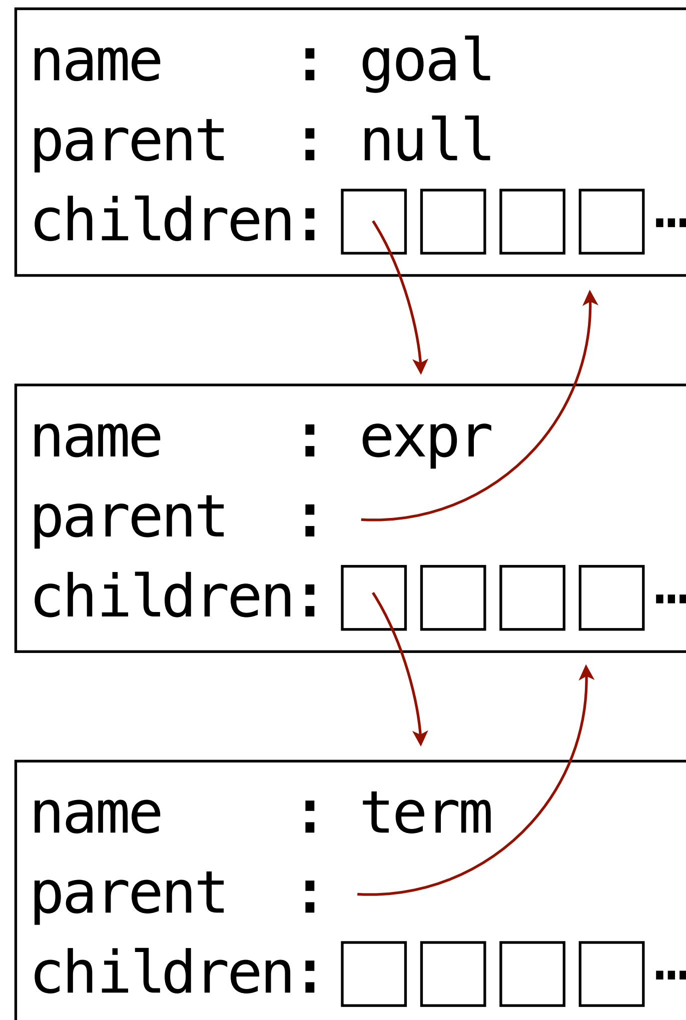
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

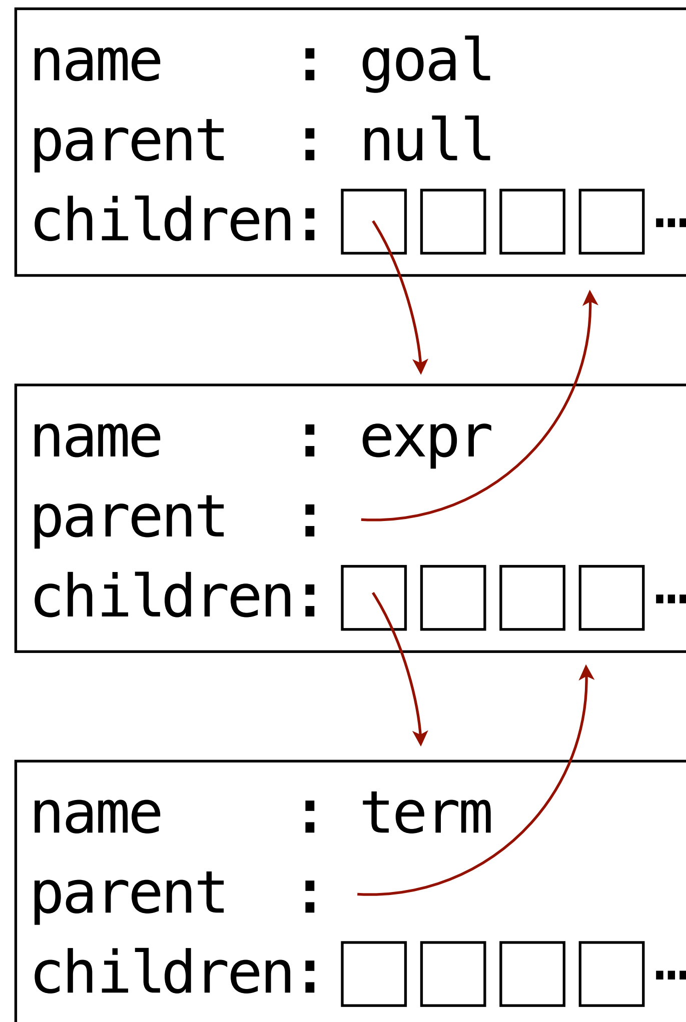
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

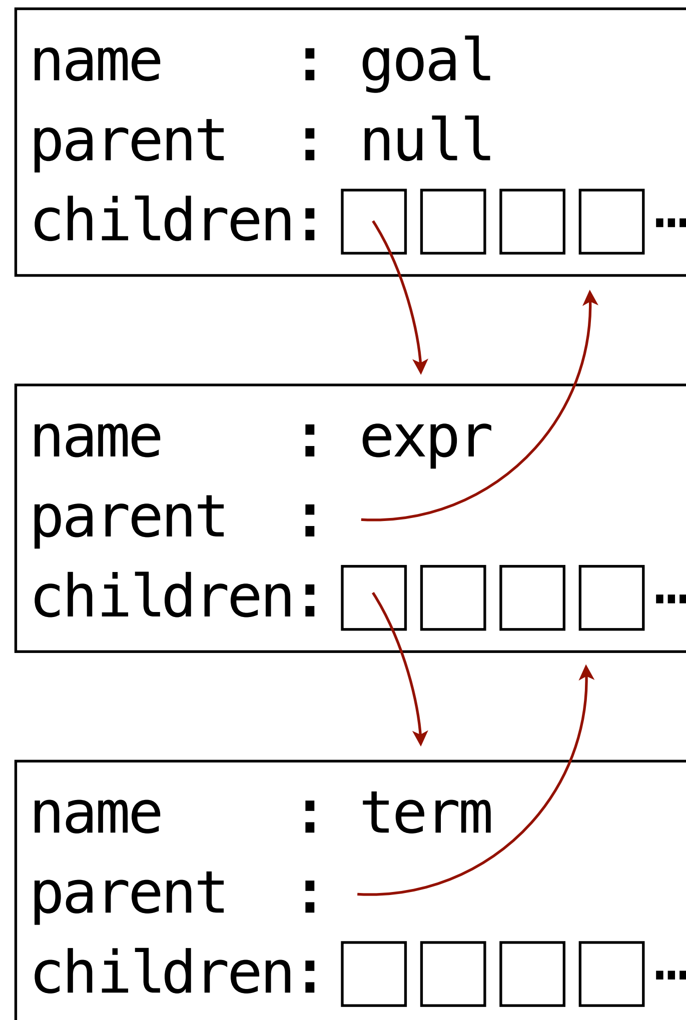
input tokens: **`a + 2`**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

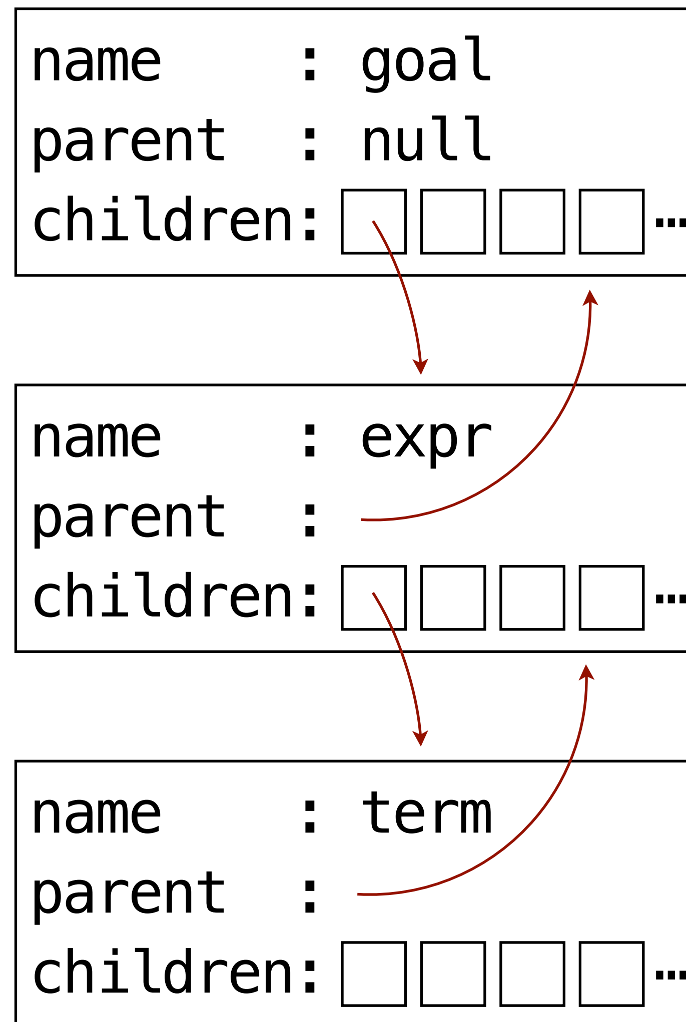
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

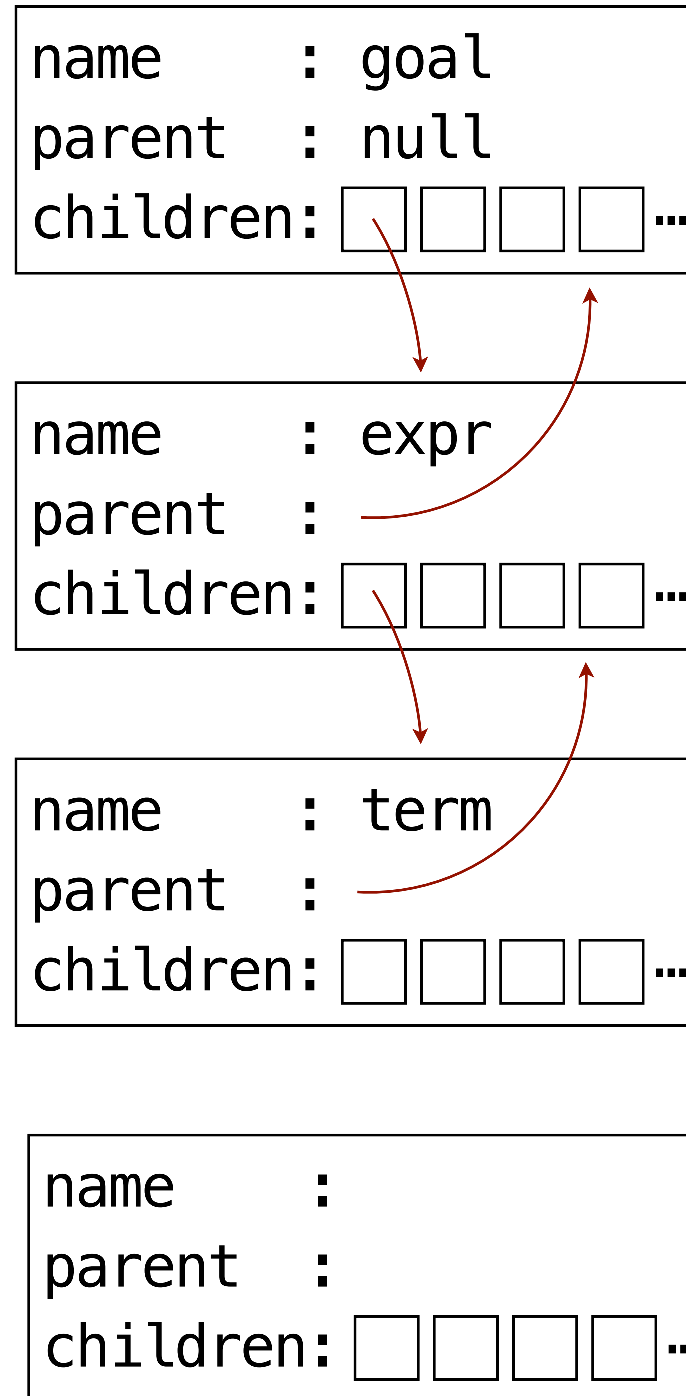
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

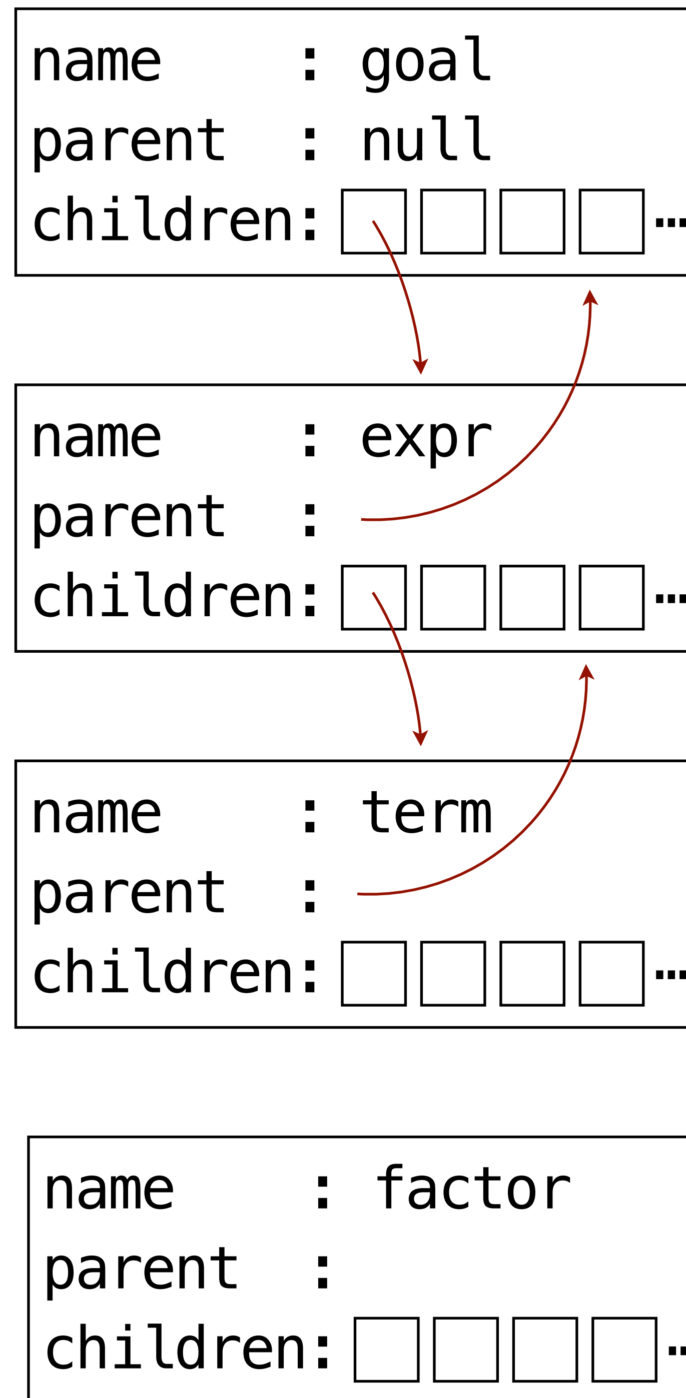
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

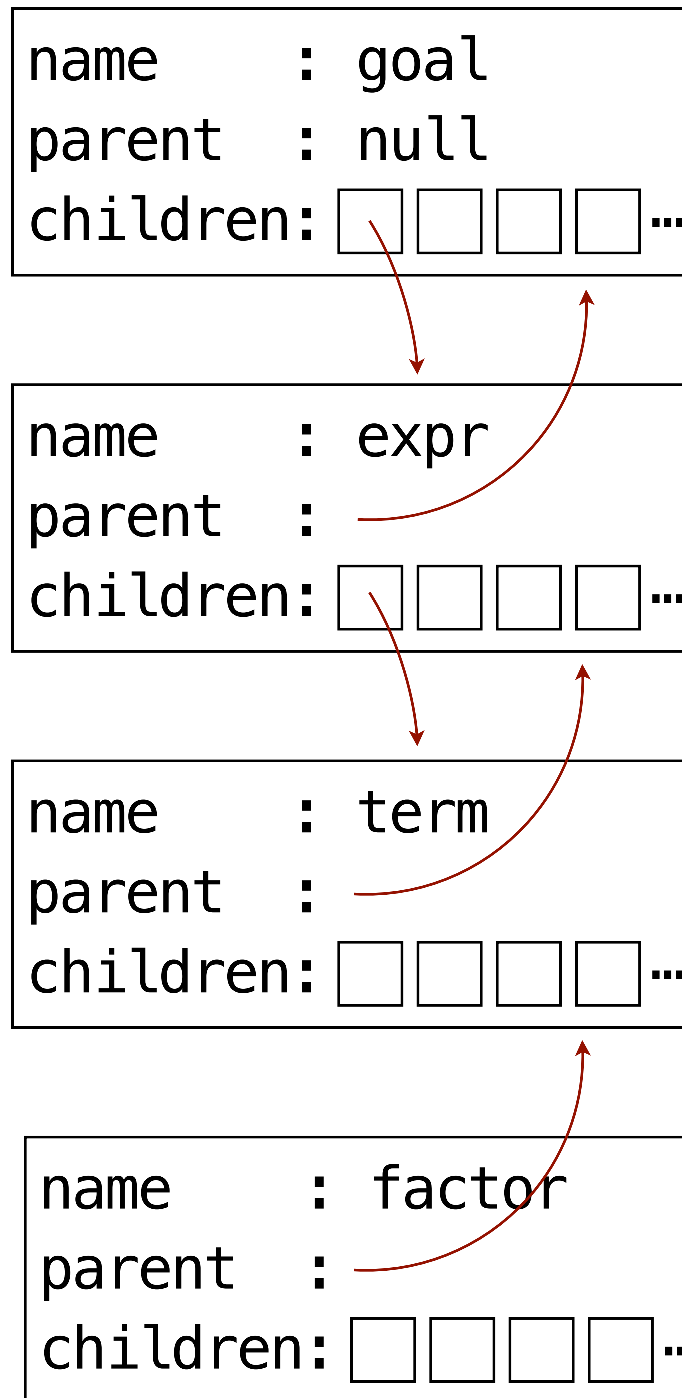
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

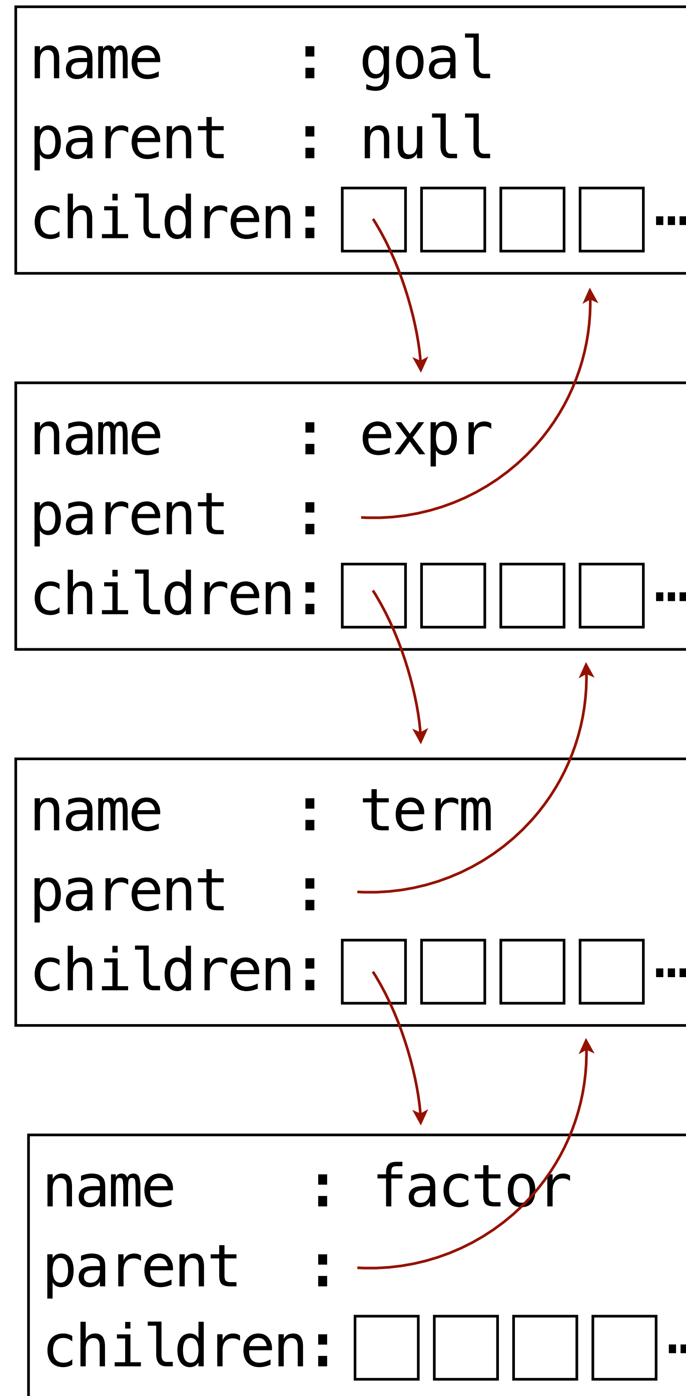
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

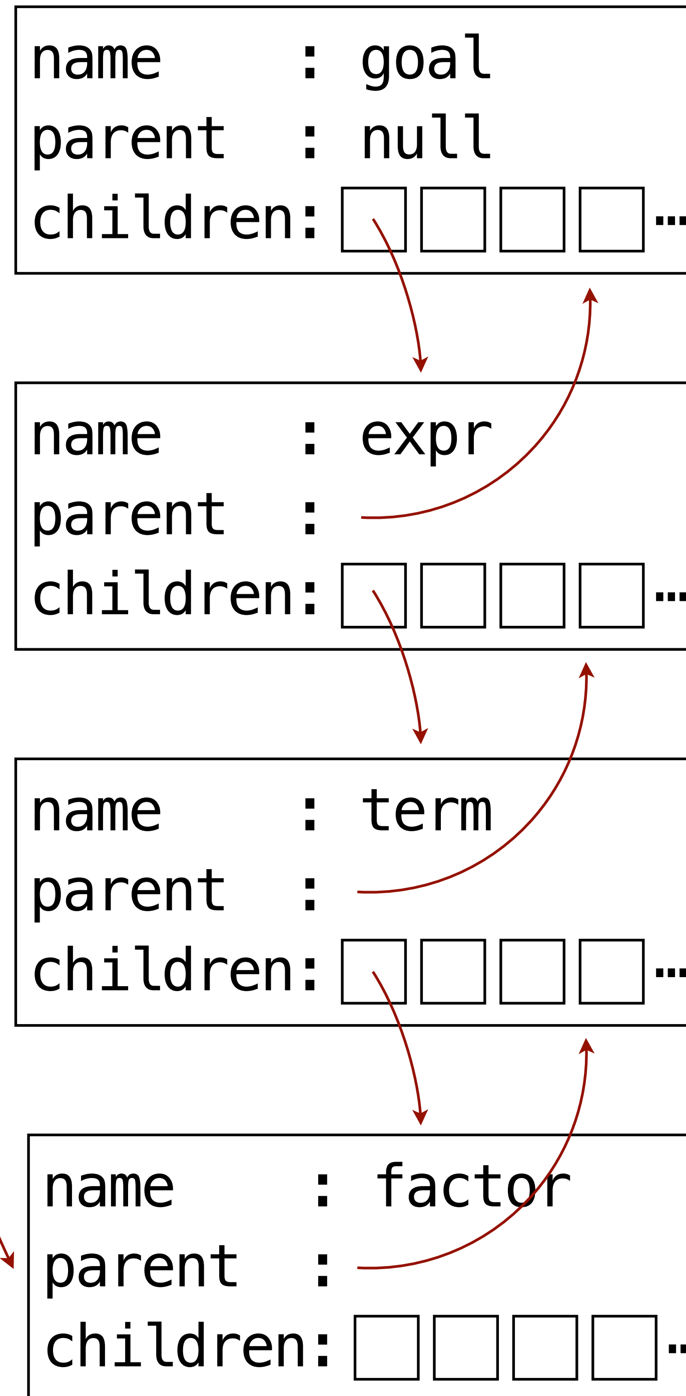
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

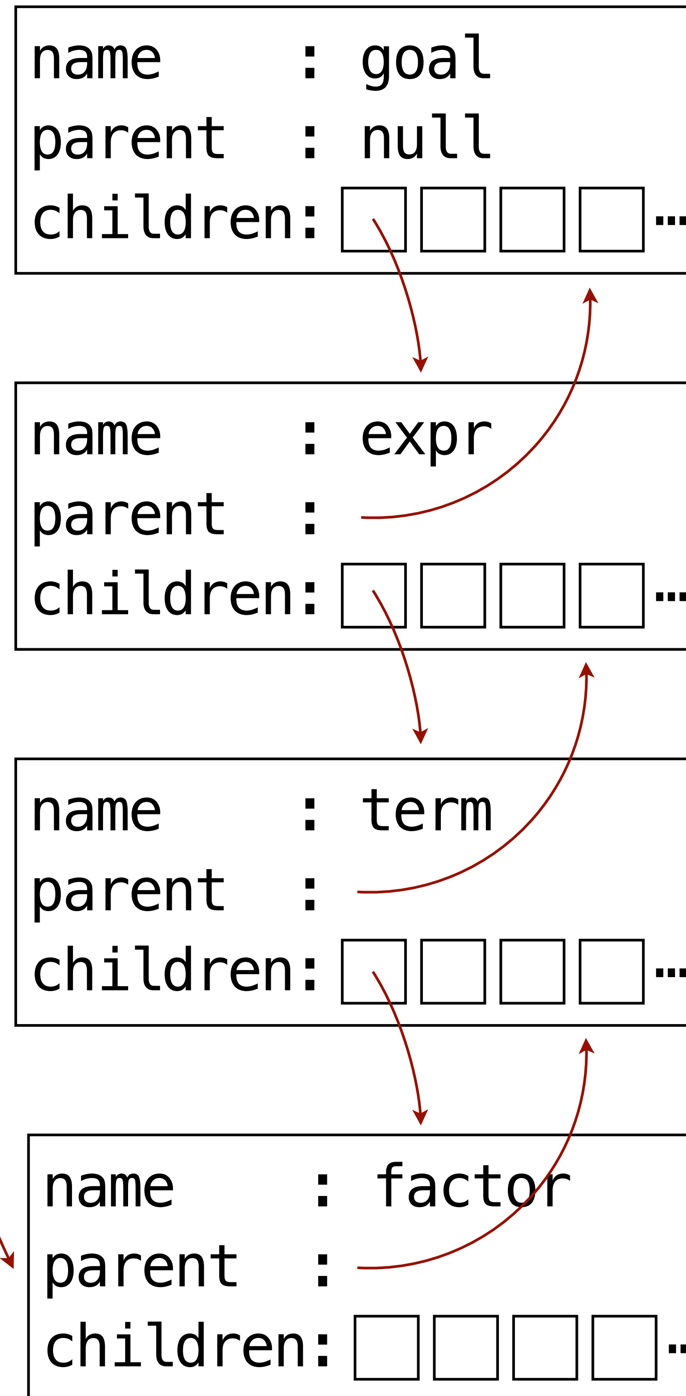
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

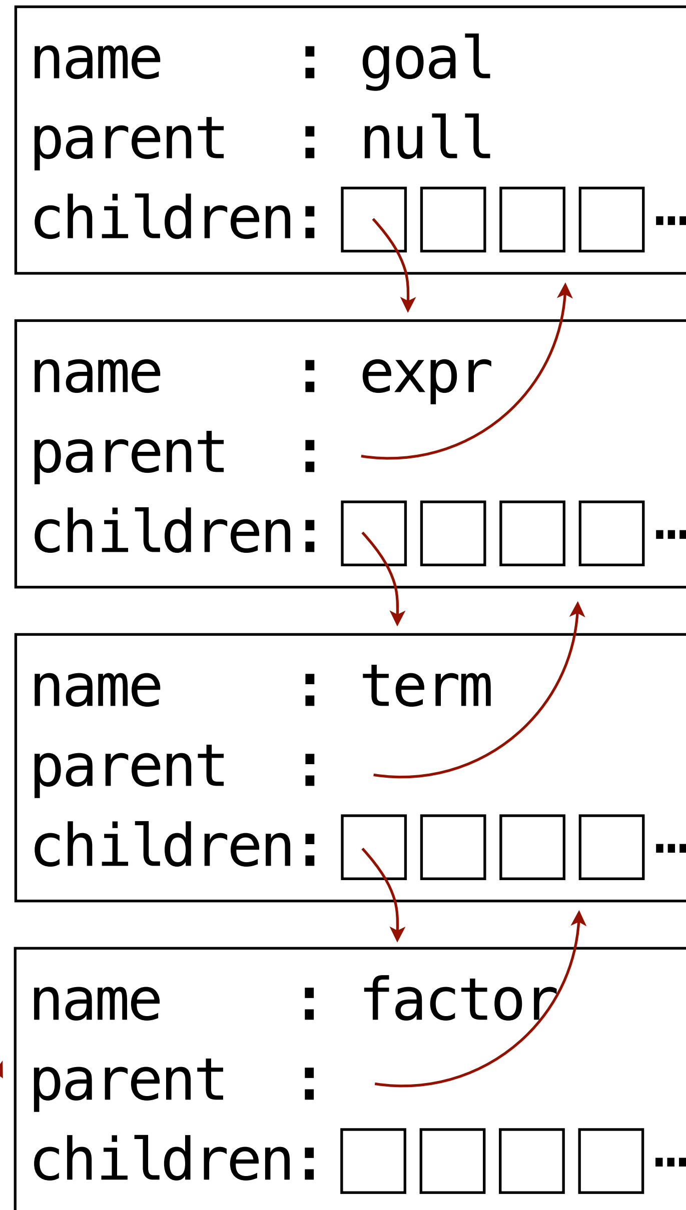
input tokens: **a + 2**

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

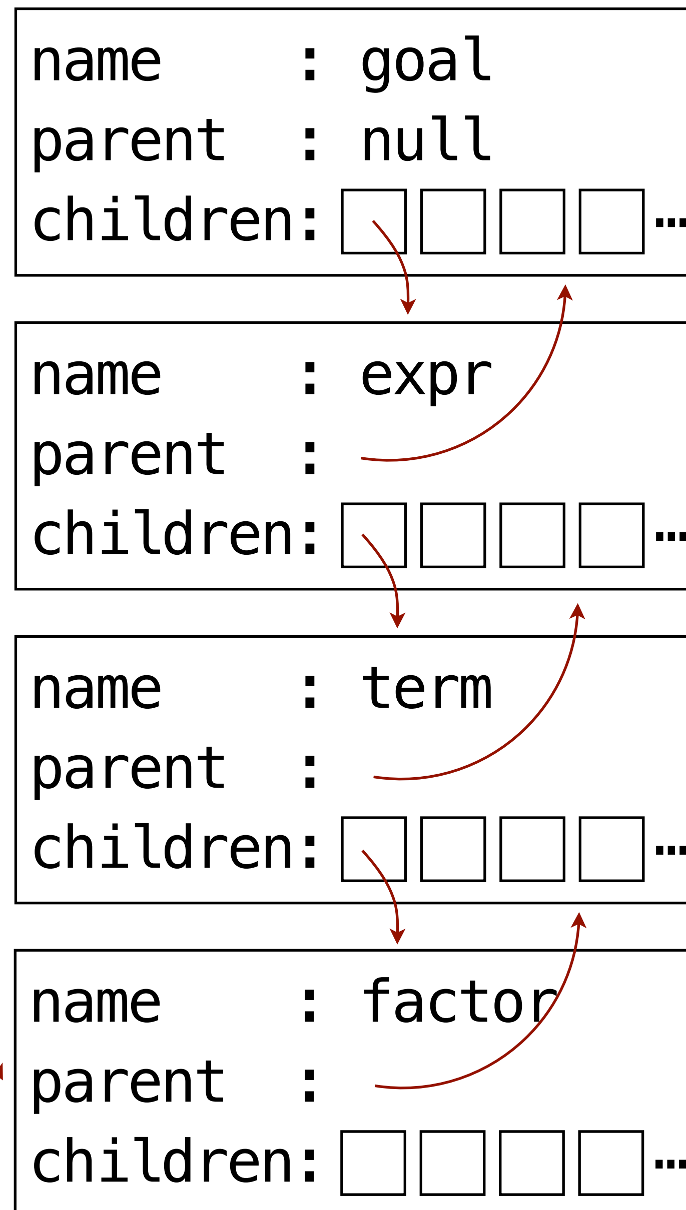
input tokens: a + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

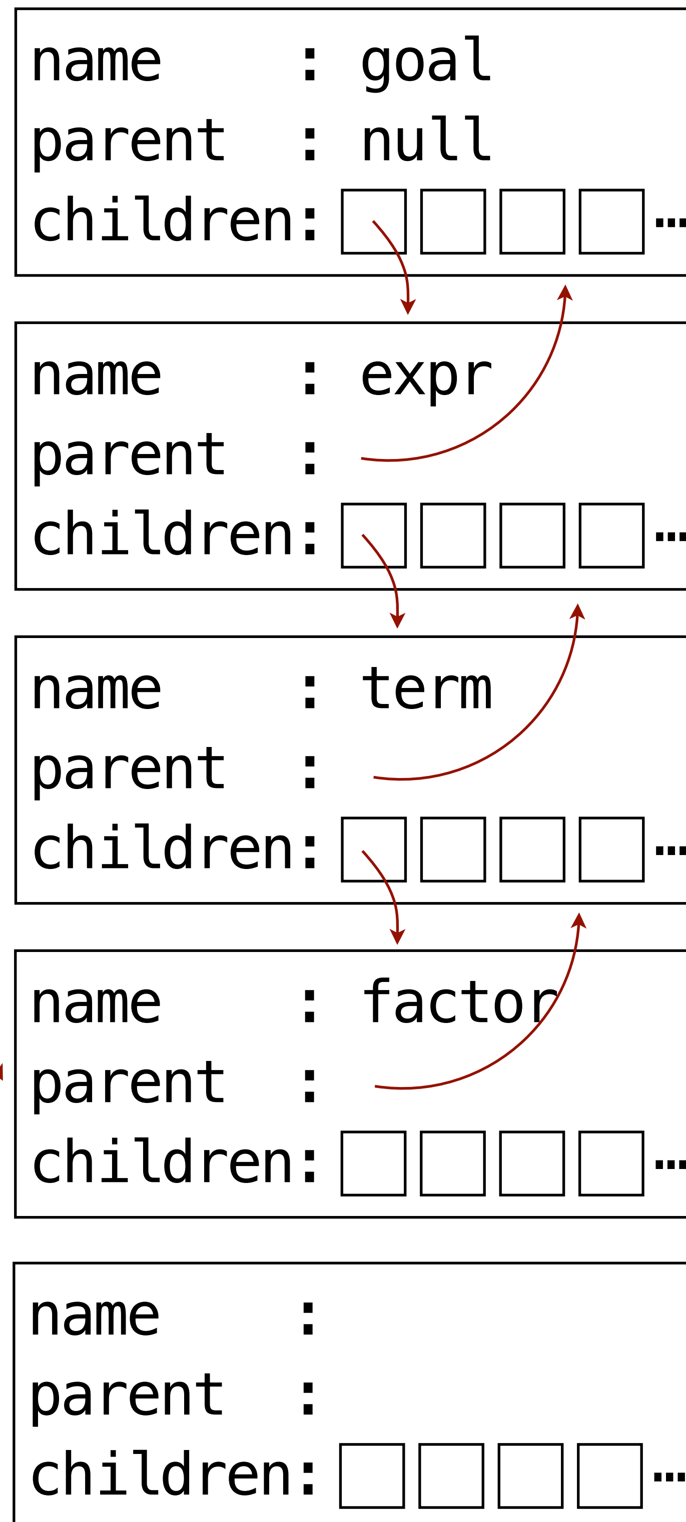
input tokens: a + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

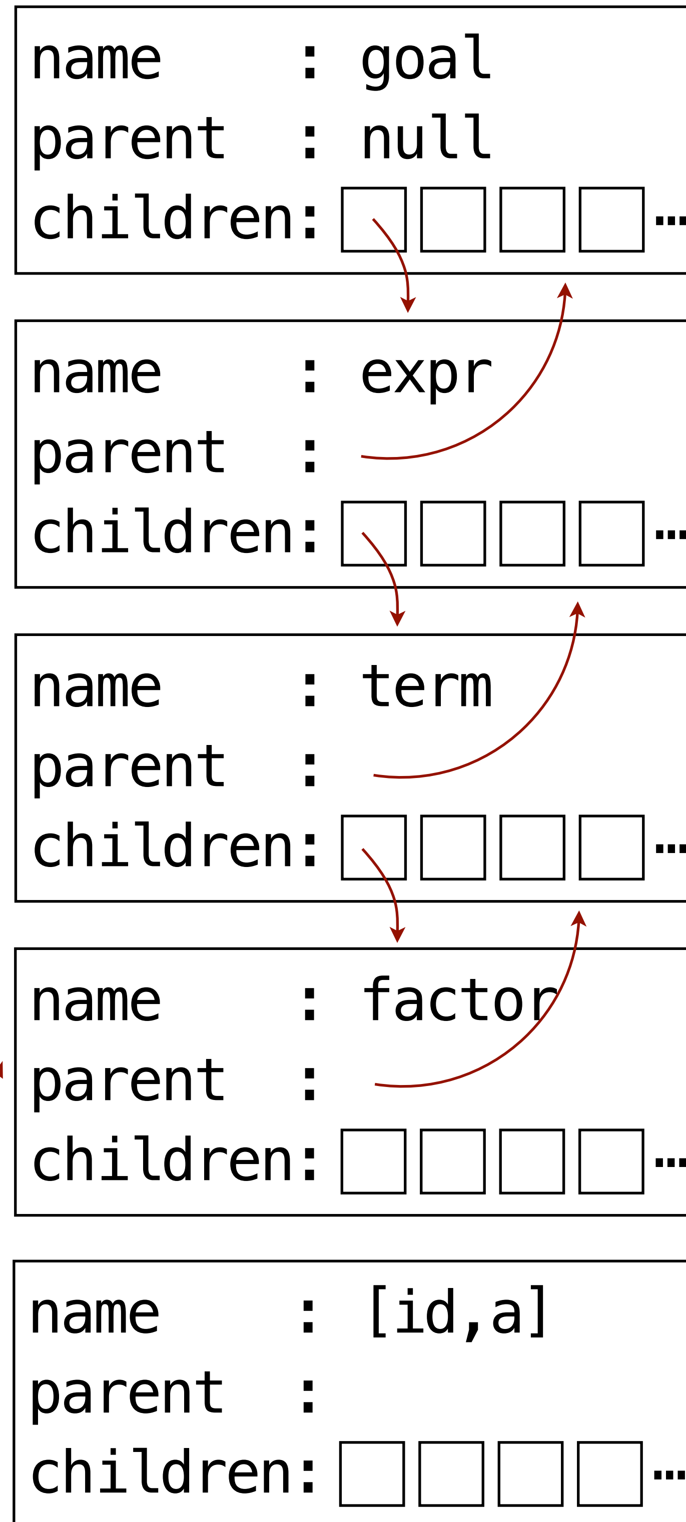
input tokens: a + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

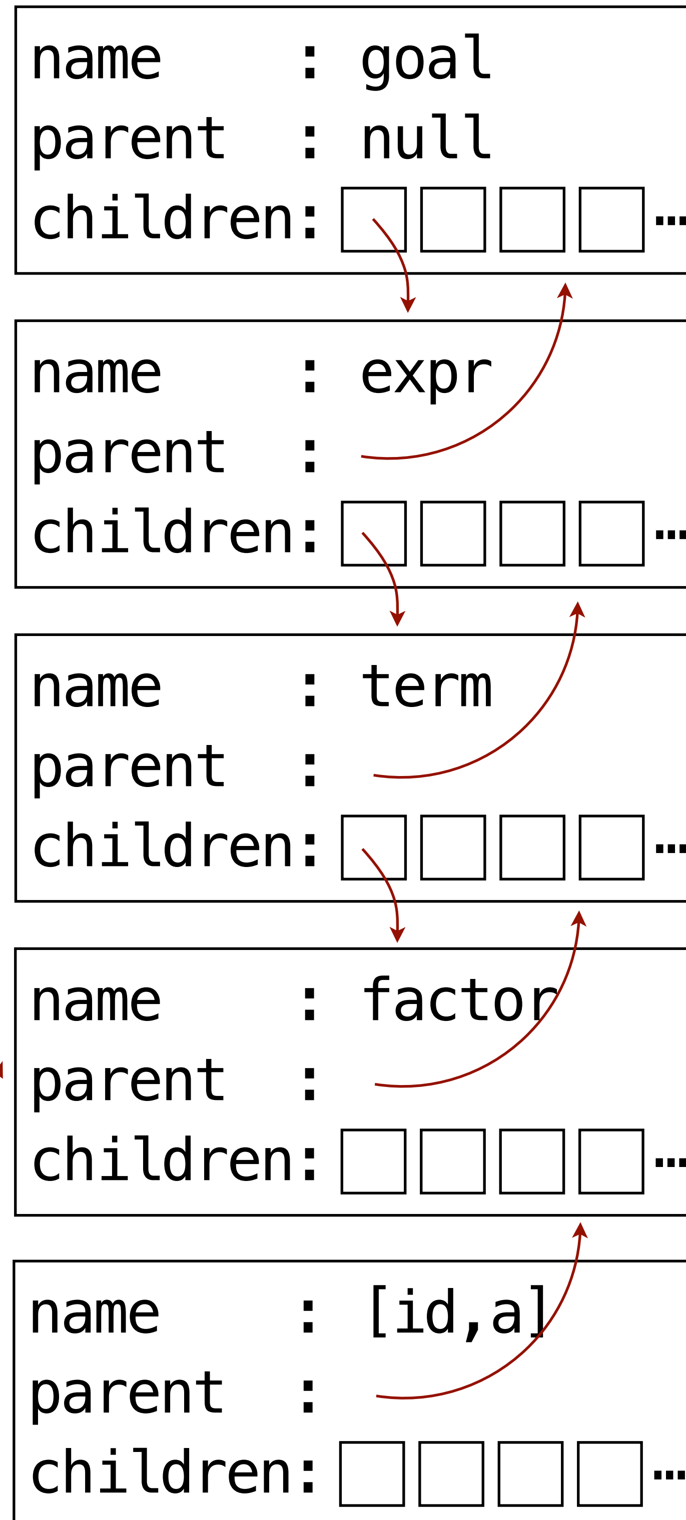
input tokens: a + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

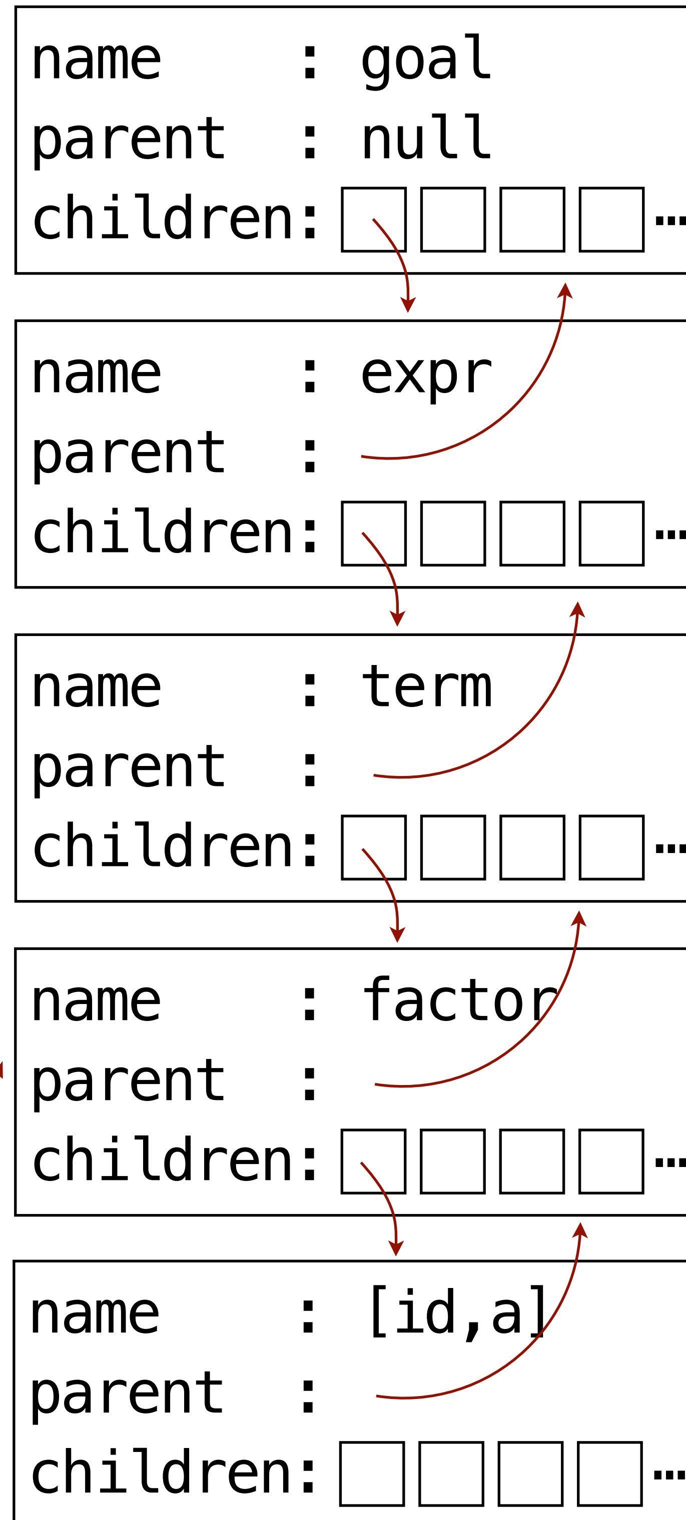
input tokens: ~~x~~ + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
```

```
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

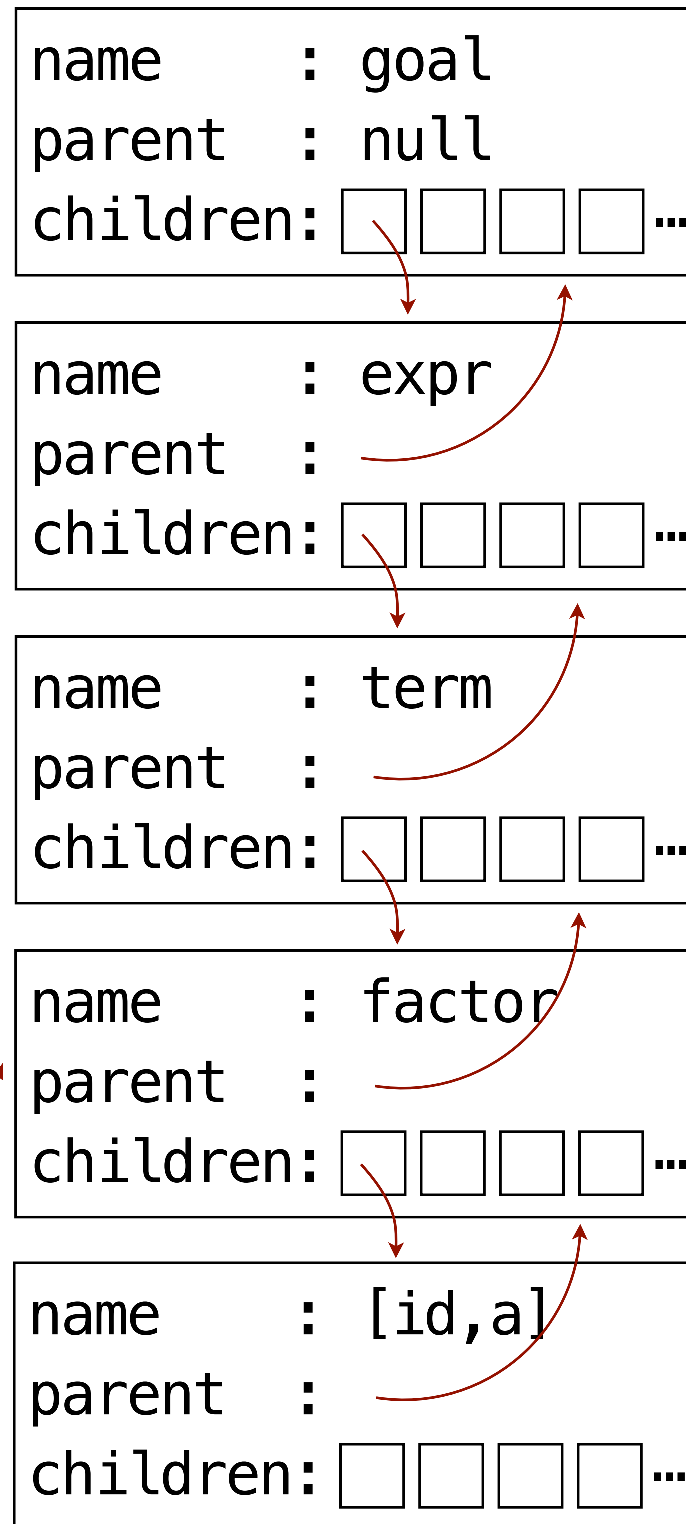
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

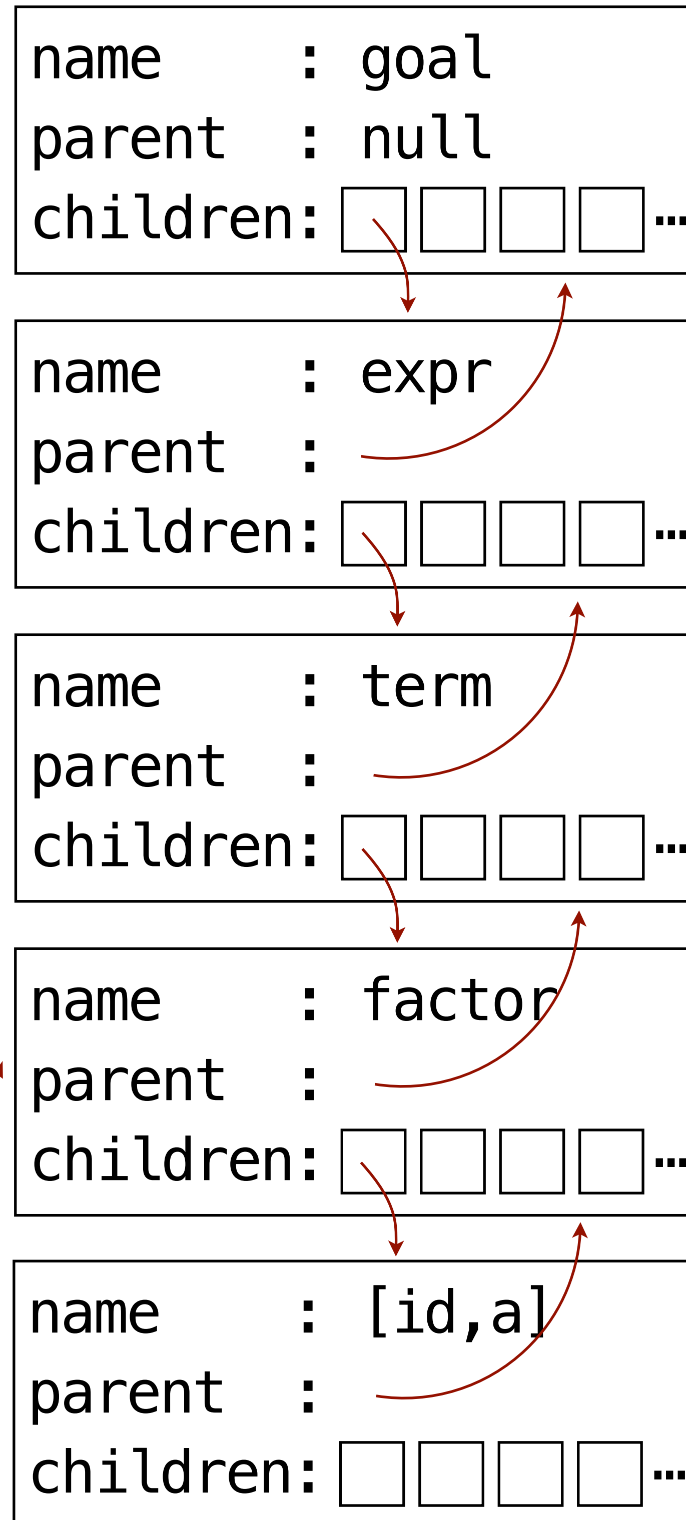
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    ?
  }
```

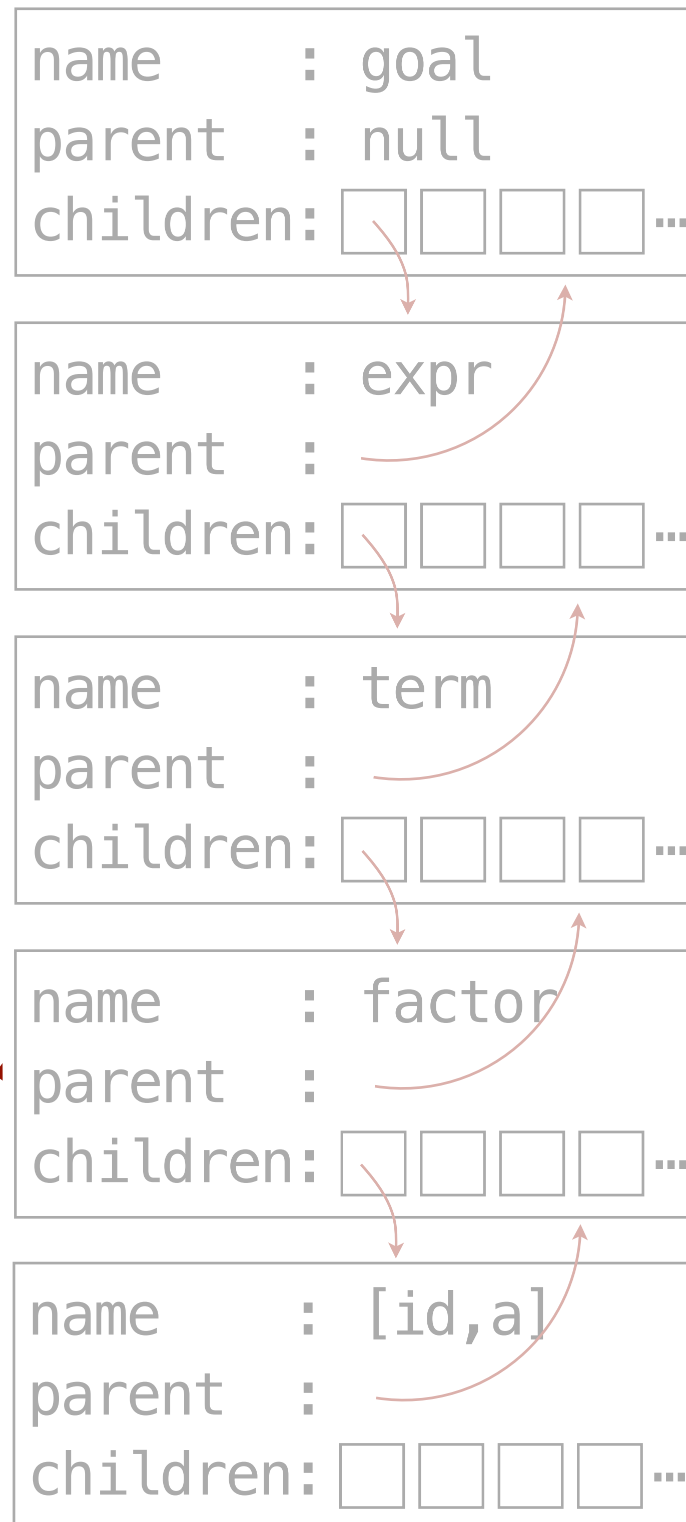



```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```



What is the mystery function?

What does it need to do?

```
this.mysteryFunction(){
  ?
}
```

input tokens: x + 2

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

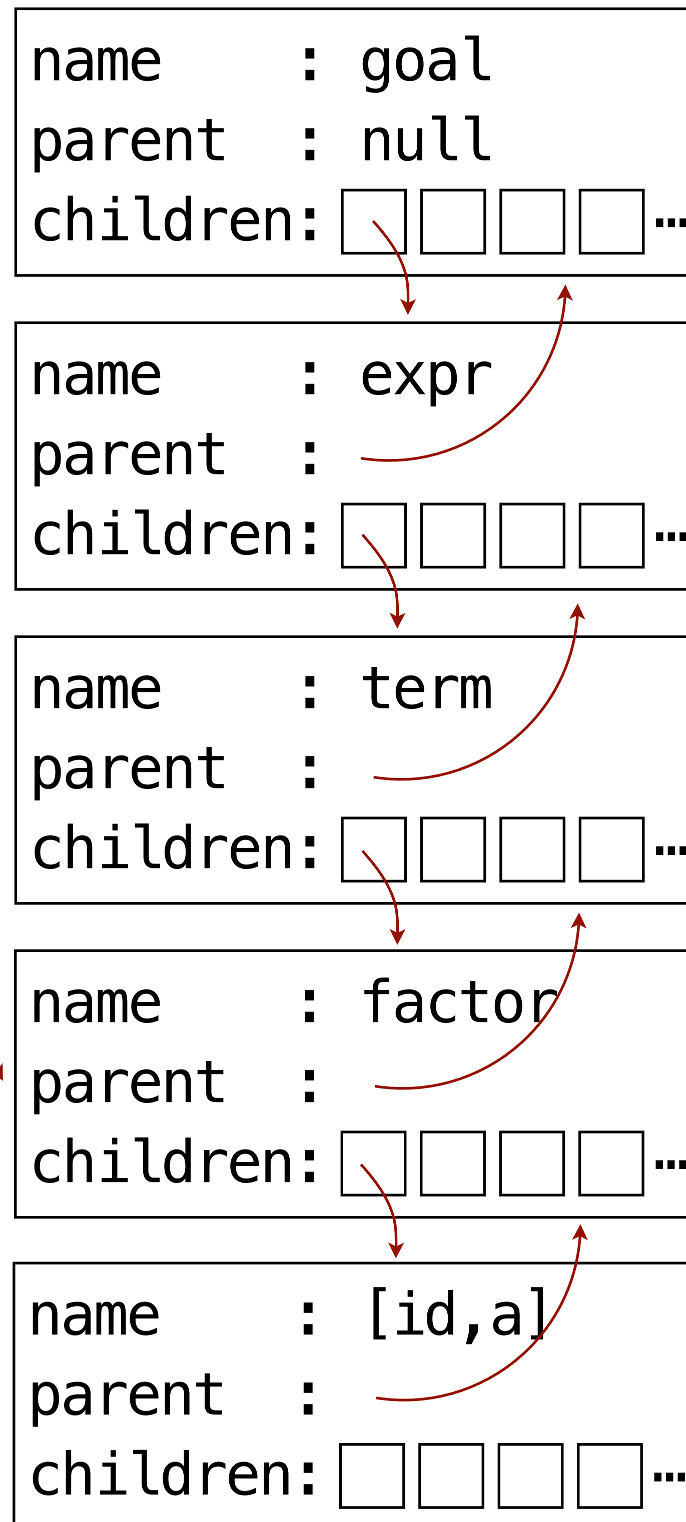
Implementing a Concrete Syntax Tree

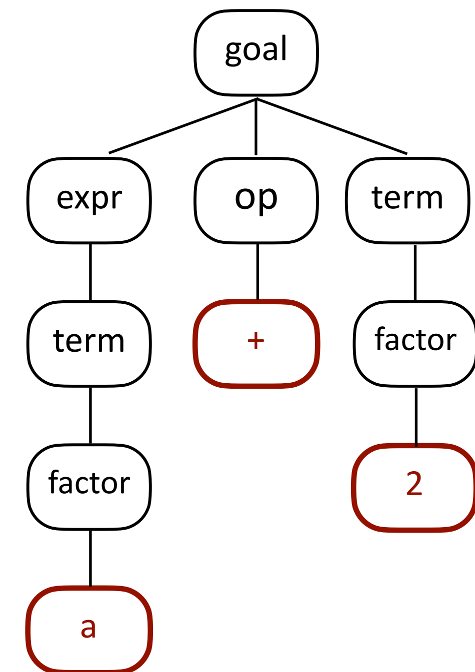
```
function Tree() =
  this.root =
  this.current =
```

What is the mystery function?

What does it need to do?

```
  this.mysteryFunction(){
    ?
  }
```





This is the CST we want. We're getting close.

The next terminal will be the + operator. But we don't want that at the bottom of the tree.

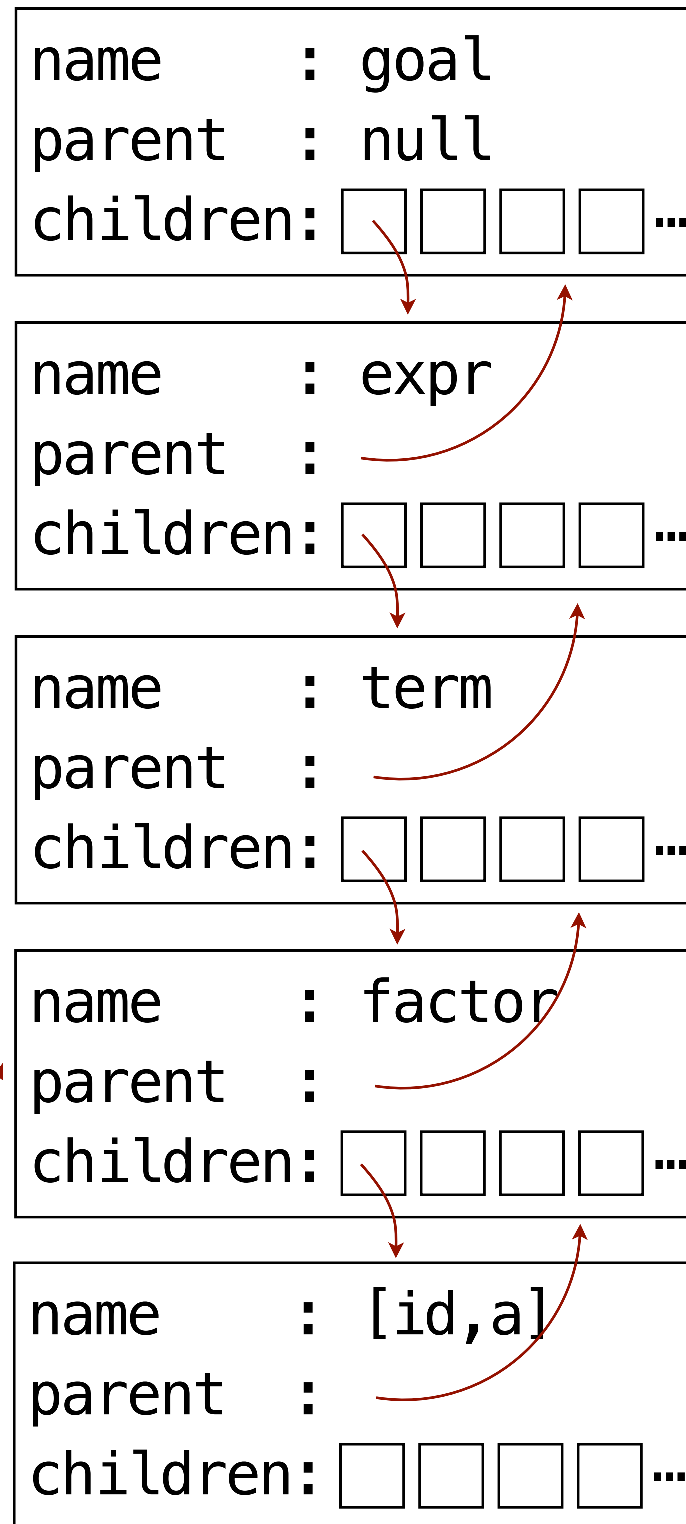
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.mysteryFunction(){
    this.current =
              this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  mystery function
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  mystery function
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

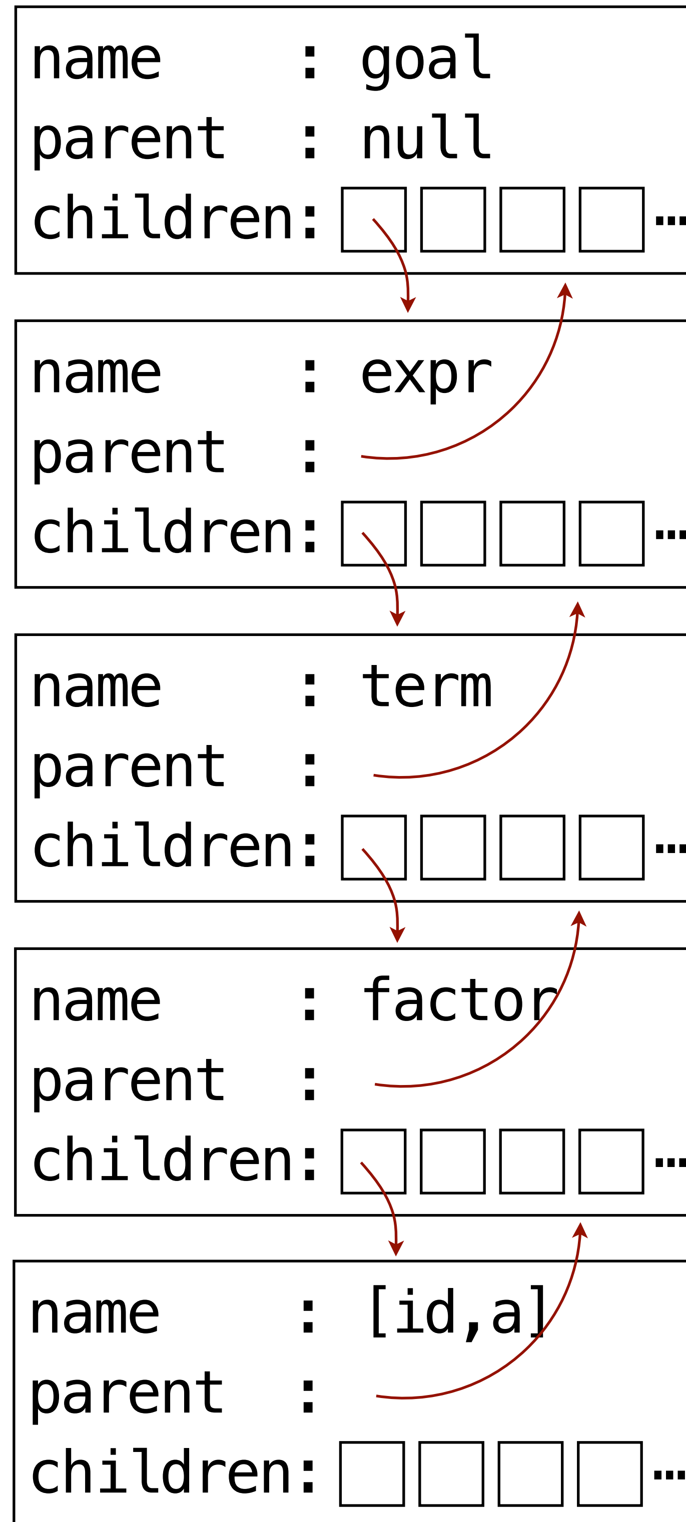
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
              this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  mystery function
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

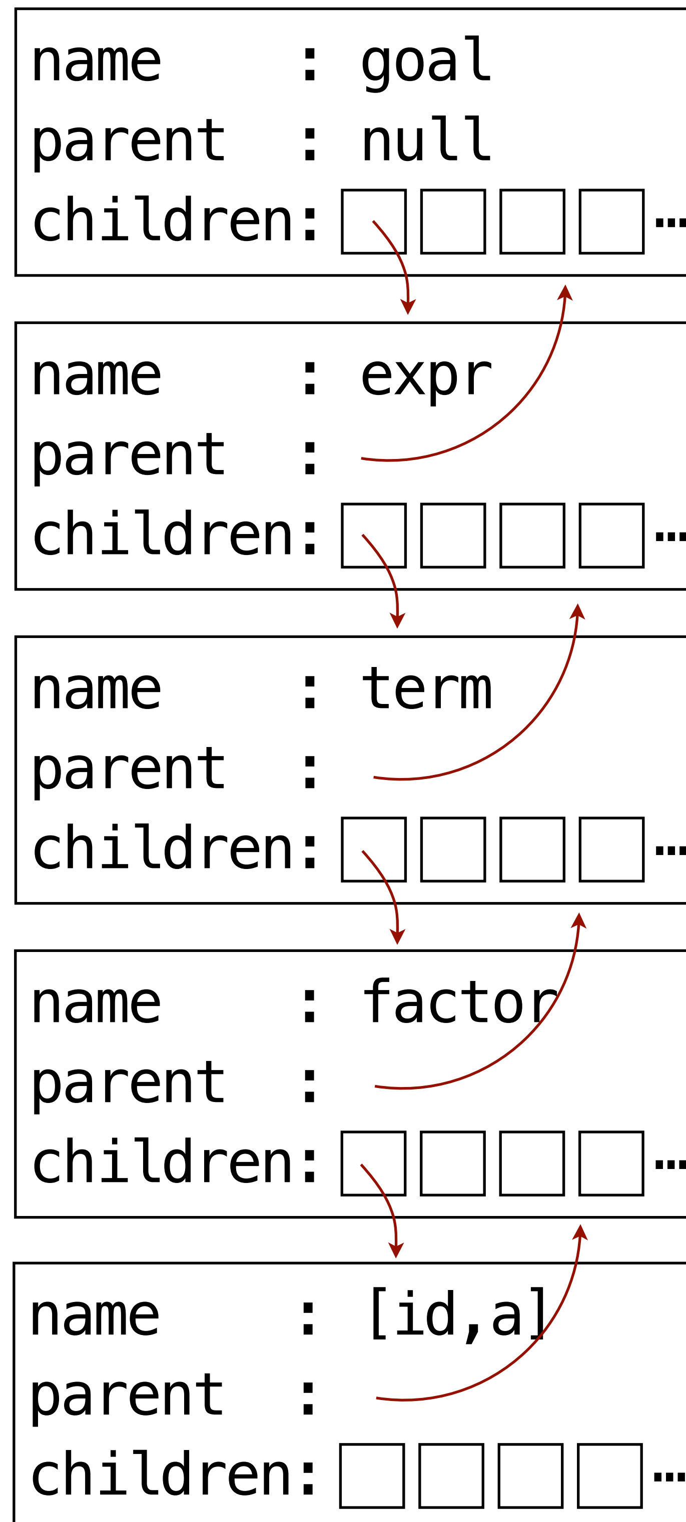
input tokens: ~~x~~ + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
             this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

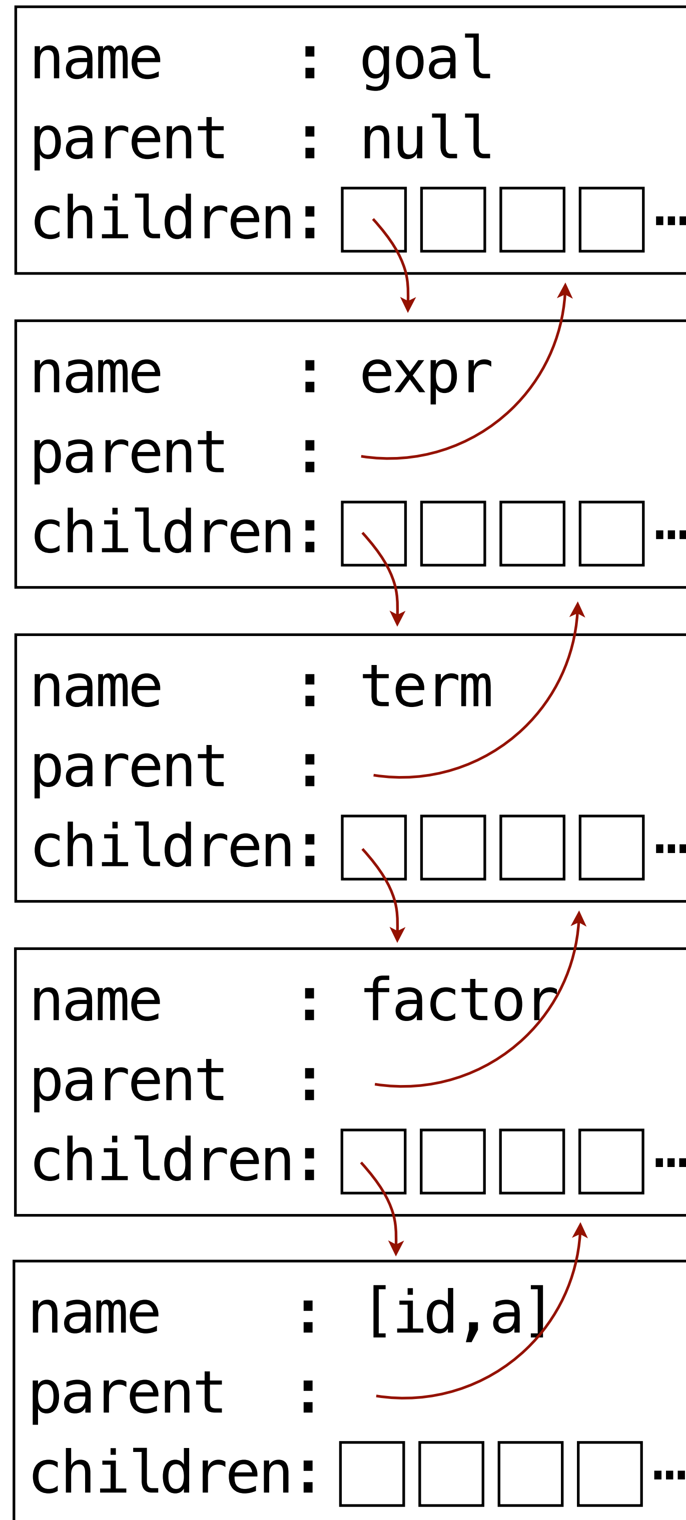
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
              this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  mystery function
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

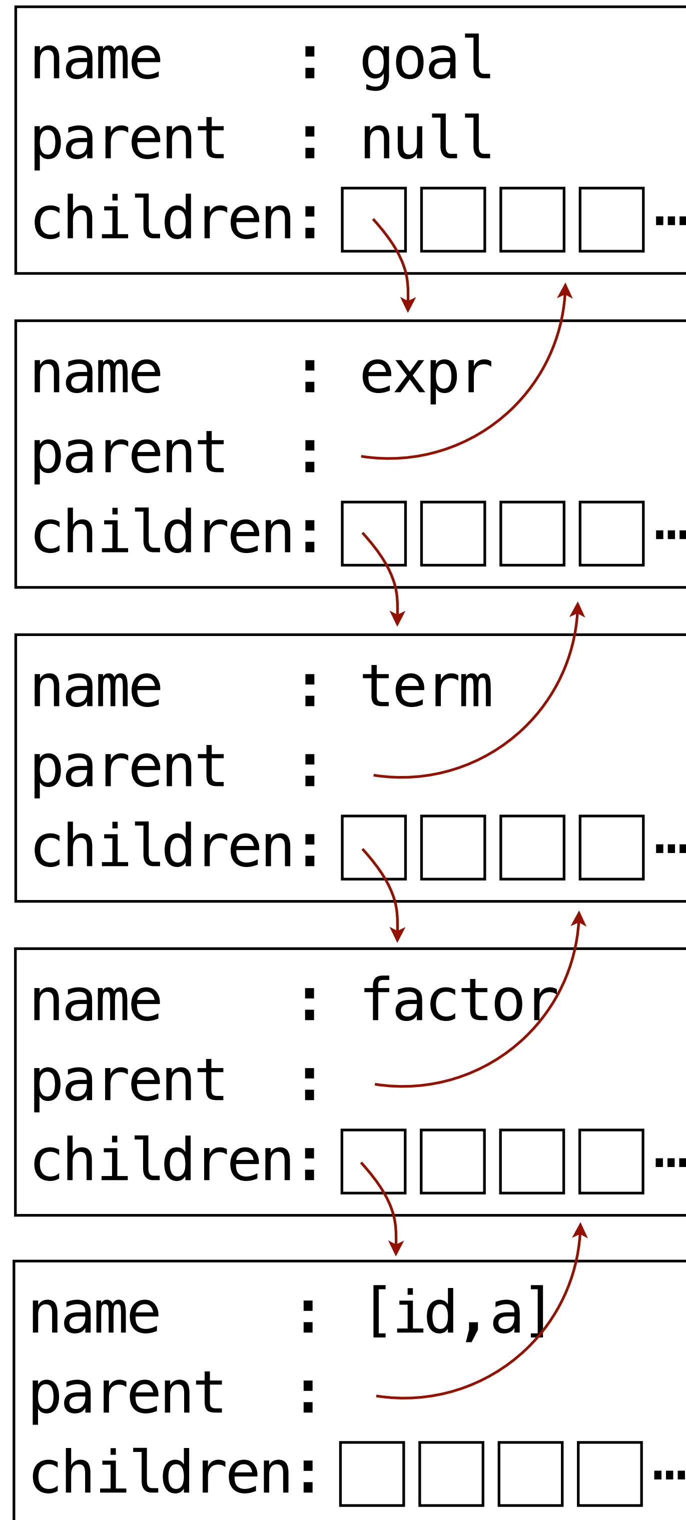
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
             this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

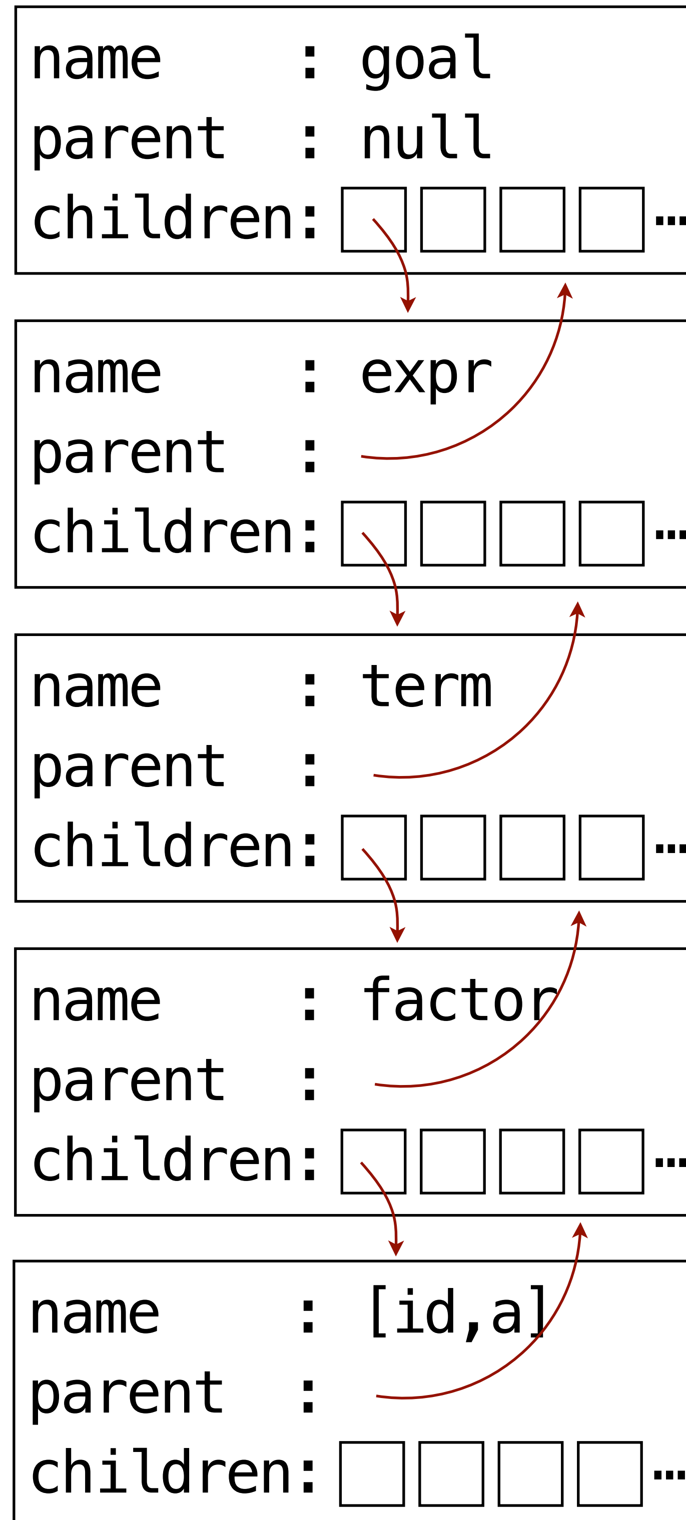
input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
            this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  mystery function
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: x + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
              this.current.parent
  }
```

```
name    : goal
parent  : null
children: [ ][ ][ ][ ]...

name    : expr
parent  :
children: [ ][ ][ ][ ]...

name    : term
parent  :
children: [ ][ ][ ][ ]...

name    : factor
parent  :
children: [ ][ ][ ][ ]...

name    : [id,a]
parent  :
children: [ ][ ][ ][ ]...
```

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  moveUp()
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

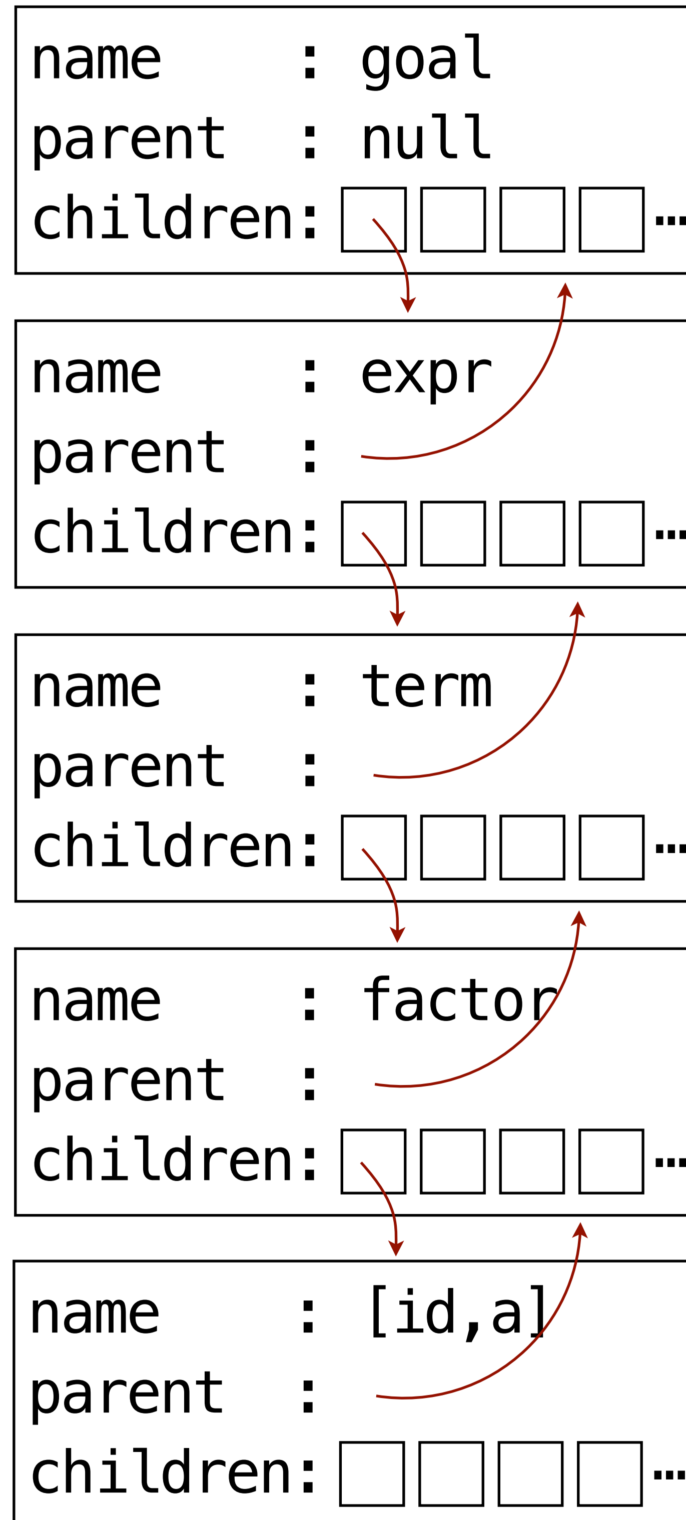
input tokens: ~~x~~ + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =

  this.addNode(kind,label){
    n = new node
    n.name = label
    if this.root == null then
      this.root = n
      n.parent = null
    else
      n.parent = current
      parent.children.add(n)
    end if
    if kind != leaf then
      this.current = n
    end if
  }

  this.moveUp(){
    this.current =
            this.current.parent
  }
```




```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  moveUp()
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

input tokens: ~~x~~ + 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

```
name     : goal
parent   : null
children: □□□□…
```

```
name     : expr
parent   :
children: □□□□…
```

```
name     : term
parent   :
children: □□□□…
```

```
name     : factor
parent   :
children: □□□□…
```

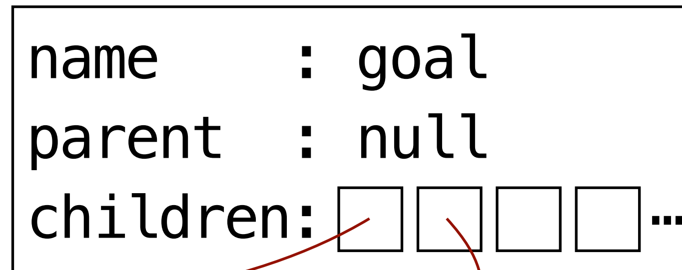
```
name     : [id,a]
parent   :
children: □□□□…
```

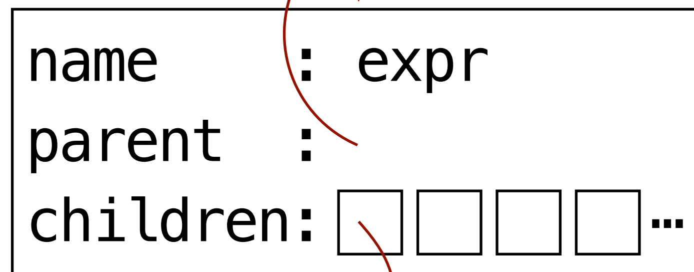
input tokens: x + 2

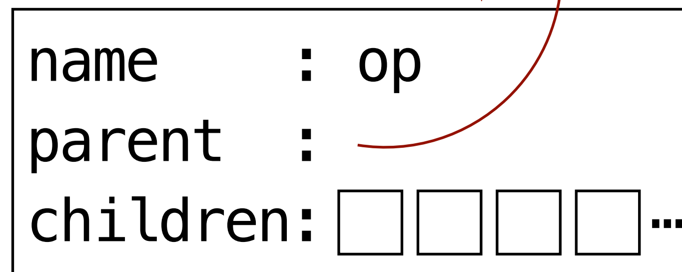
```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  moveUp()
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

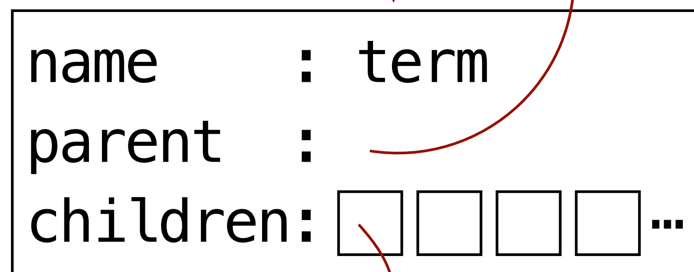
Implementing a Concrete Syntax Tree

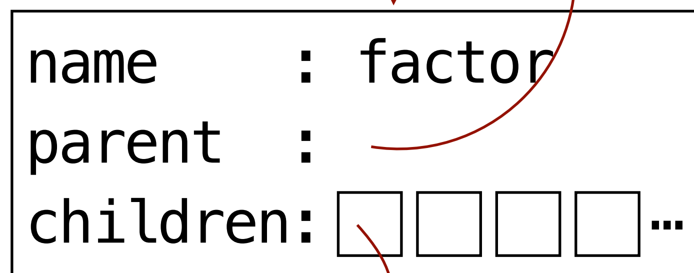
```
function Tree() =
  this.root =
  this.current =
```

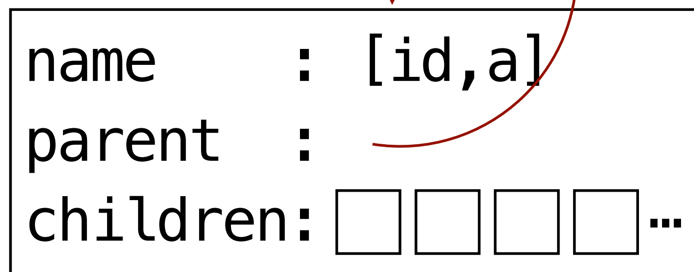












input tokens: x + 2

```
parseGoal(){
   addNode(root,goal)
   parseExpr()
   parseOp()
   parseTerm()
   moveUp()
}
parseExpr(){
   addNode(branch,expr)
   parseTerm()
   moveUp()
}
parseTerm(){
   addNode(branch,term)
   parseFactor()
   moveUp()
}
parseFactor(){
   addNode(branch,factor)
   match({num,id})
   moveUp()
}
parseOp(){
   addNode(branch,op)
   match({+,-})
   moveUp()
}
match(expected){
   x = checkExpected
   if x addNode(leaf,x)
   else error
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

```
name     : goal
parent   : null
children: □□□□…
```

```
name     : expr
parent   :
children: □□□□…
```

```
name     : op
parent   :
children: □□□□…
```

```
name     : term
parent   :
children: □□□□…
```

```
name     : factor
parent   :
children: □□□□…
```

```
name     : [id,a]
parent   :
children: □□□□…
```

input tokens: x + 2

```
parseGoal(){
   addNode(root,goal)
   parseExpr()
   parseOp()
   parseTerm()
   moveUp()
}
parseExpr(){
   addNode(branch,expr)
   parseTerm()
   moveUp()
}
parseTerm(){
   addNode(branch,term)
   parseFactor()
   moveUp()
}
parseFactor(){
   addNode(branch,factor)
   match({num,id})
   moveUp()
}
parseOp(){
   addNode(branch,op)
   match({+,-})
   moveUp()
}
match(expected){
   x = checkExpected
   if x addNode(leaf,x)
   else error
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

name : goal
parent : null
children: ☐☐☐☐…

name : expr
parent :
children: ☐☐☐☐…

name : op
parent :
children: ☐☐☐☐…

name : term
parent :
children: ☐☐☐☐…

name : +
parent :
children: ☐☐☐☐…

name : factor
parent :
children: ☐☐☐☐…

name : [id,a]
parent :
children: ☐☐☐☐…

input tokens: x x 2

```
parseGoal(){
   addNode(root,goal)
   parseExpr()
   parseOp()
   parseTerm()
   moveUp()
}
parseExpr(){
   addNode(branch,expr)
   parseTerm()
   moveUp()
}
parseTerm(){
   addNode(branch,term)
   parseFactor()
   moveUp()
}
parseFactor(){
   addNode(branch,factor)
   match({num,id})
   moveUp()
}
parseOp(){
   addNode(branch,op)
   match({+,-})
   moveUp()
}
match(expected){
   x = checkExpected
   if x addNode(leaf,x)
   else error
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

```
name     : goal
parent   : null
children: □□□□…
```

```
name     : expr
parent   :
children: □□□□…
```

```
name     : op
parent   :
children: □□□□…
```

```
name     : term
parent   :
children: □□□□…
```

```
name     : +
parent   :
children: □□□□…
```

```
name     : factor
parent   :
children: □□□□…
```

```
name     : [id,a]
parent   :
children: □□□□…
```

input tokens: ~~x~~ ~~x~~ 2

```
parseGoal(){
  addNode(root,goal)
  parseExpr()
  parseOp()
  parseTerm()
  moveUp()
}
parseExpr(){
  addNode(branch,expr)
  parseTerm()
  moveUp()
}
parseTerm(){
  addNode(branch,term)
  parseFactor()
  moveUp()
}
parseFactor(){
  addNode(branch,factor)
  match({num,id})
  moveUp()
}
parseOp(){
  addNode(branch,op)
  match({+,-})
  moveUp()
}
match(expected){
  x = checkExpected
  if x addNode(leaf,x)
  else error
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

```
name     : goal
parent   : null
children: □□□□…
```

```
name     : expr
parent   :
children: □□□□…
```

```
name     : op
parent   :
children: □□□□…
```

```
name     : term
parent   :
children: □□□□…
```

```
name     : +
parent   :
children: □□□□…
```

```
name     : factor
parent   :
children: □□□□…
```

```
name     : [id,a]
parent   :
children: □□□□…
```

input tokens: ~~x~~ ~~x~~ 2

```
parseGoal(){
   addNode(root,goal)
   parseExpr()
   parseOp()
   parseTerm()
   moveUp()
}
parseExpr(){
   addNode(branch,expr)
   parseTerm()
   moveUp()
}
parseTerm(){
   addNode(branch,term)
   parseFactor()
   moveUp()
}
parseFactor(){
   addNode(branch,factor)
   match({num,id})
   moveUp()
}
parseOp(){
   addNode(branch,op)
   match({+,−})
   moveUp()
}
match(expected){
   x = checkExpected
   if x addNode(leaf,x)
   else error
}
```

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

```
name    : goal
parent  : null
children: □□□□…
```

```
name    : expr
parent  :
children: □□□□…
```

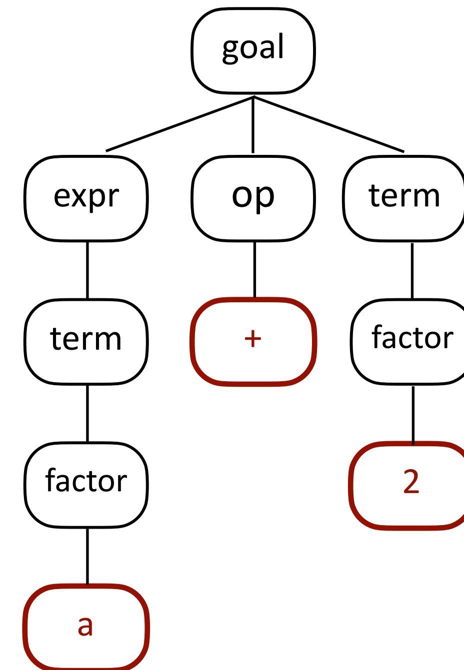
```
name    : op
parent  :
children: □□□□…
```

```
name    : term
parent  :
children: □□□□…
```

```
name    : +
parent  :
children: □□□□…
```

```
name    : factor
parent  :
children: □□□□…
```

```
name    : [id,a]
parent  :
children: □□□□…
```



This is the CST we want.
We're getting closer.

input tokens: x x 2

Implementing a Concrete Syntax Tree

```
function Tree() =
  this.root =
  this.current =
```

name : goal
parent : null
children: ☐☐☐☐…

name : expr
parent :
children: ☐☐☐☐…

name : op
parent :
children: ☐☐☐☐…

name : term
parent :
children: ☐☐☐☐…

name : term
parent :
children: ☐☐☐☐…

name : +
parent :
children: ☐☐☐☐…

name : factor
parent :
children: ☐☐☐☐…

name : factor
parent :
children: ☐☐☐☐…

name : [num,2]
parent :
children: ☐☐☐☐…

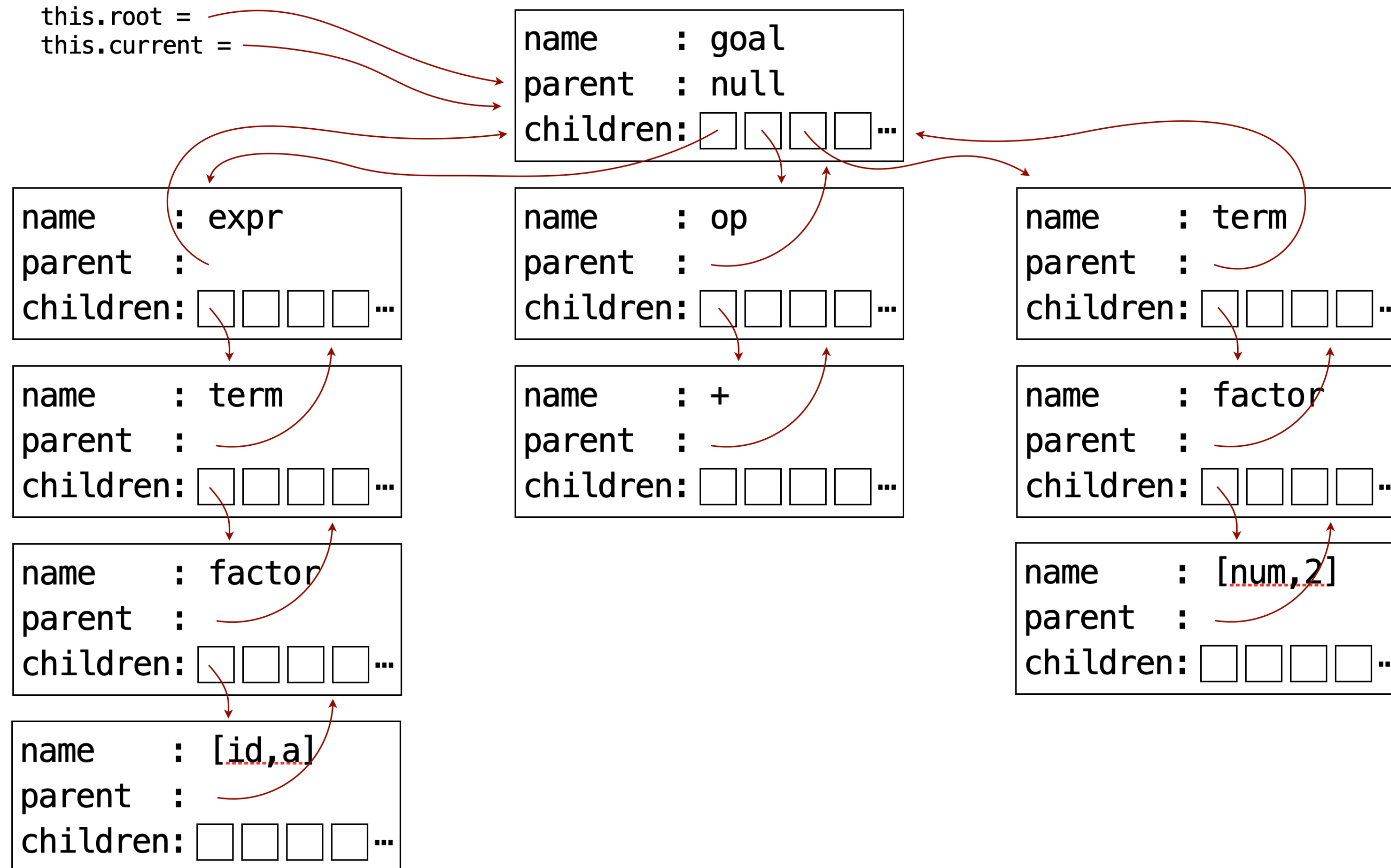
name : [id,a]
parent :
children: ☐☐☐☐…

input tokens: x x x

Implementing a Concrete Syntax Tree

input tokens: **a + 2**

CST



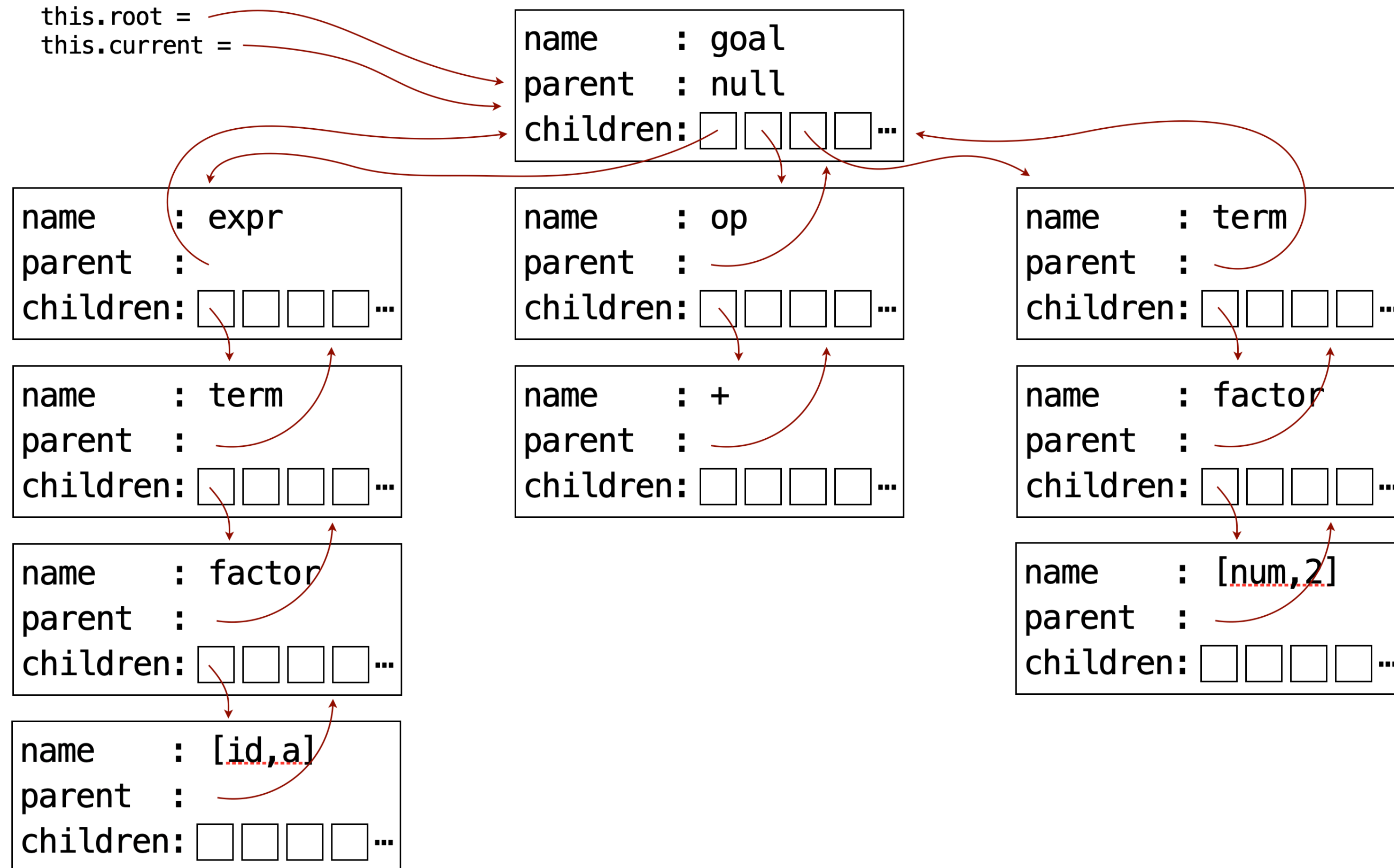
I'm a little worried about this last call to moveUp().

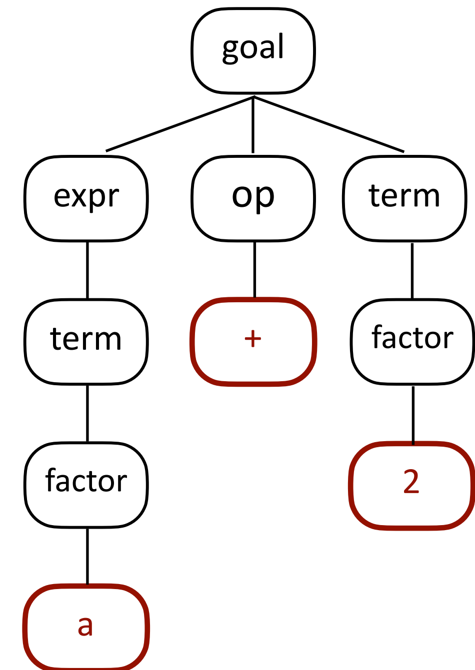
```
parseGoal(){
   addNode(root,goal)
   parseExpr()
   parseOp()
   parseTerm()
   moveUp()
}
parseExpr(){
   addNode(branch,expr)
   parseTerm()
   moveUp()
}
parseTerm(){
   addNode(branch,term)
   parseFactor()
   moveUp()
}
parseFactor(){
   addNode(branch,factor)
   match({num,id})
   moveUp()
}
parseOp(){
   addNode(branch,op)
   match({+,-})
   moveUp()
}
match(expected){
   x = checkExpected
   if x addNode(leaf,x)
   else error
}
```

Implementing a Concrete Syntax Tree

input tokens: **`a + 2`**

CST





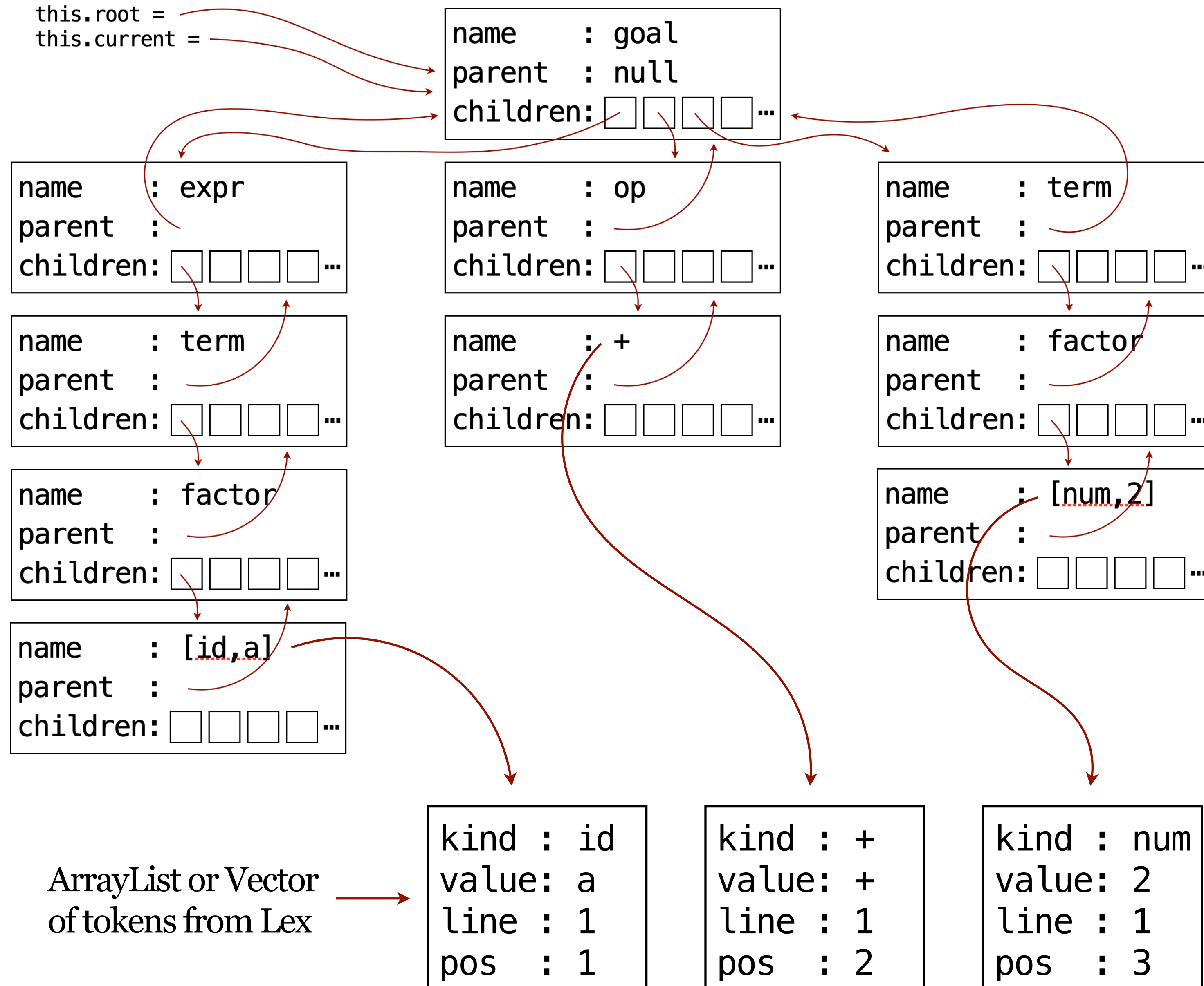
This is the CST we want.

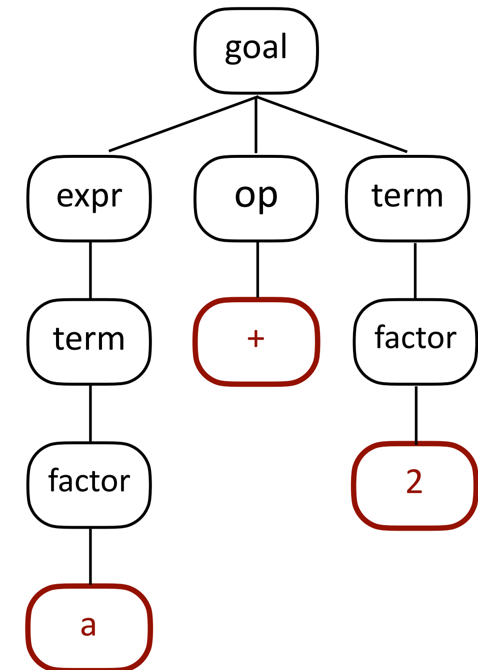
And this is the CST we've got. Very cool.

Implementing a Concrete Syntax Tree

input tokens: `a + 2`

CST





We could use references in the leaf nodes to point to tokens in the token stream we got from Lex.

We could also use clones or deep copies.

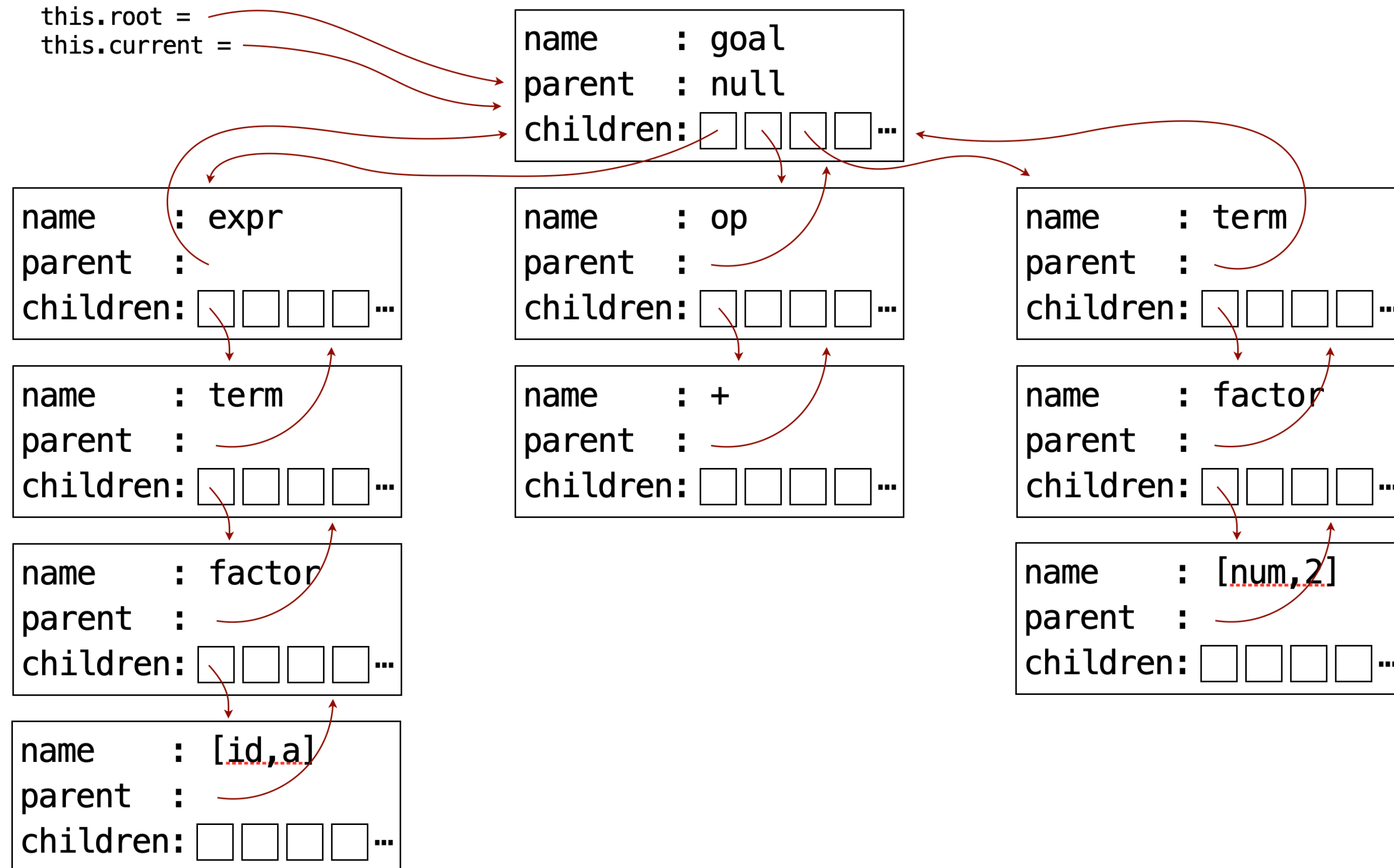
ArrayList or Vector of tokens from Lex →

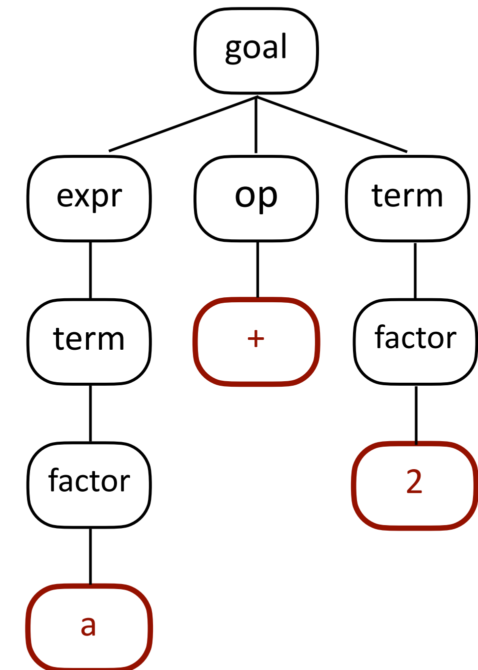
kind : id	kind : +	kind : num		
value: a	value: +	value: 2		
line : 1	line : 1	line : 1		
pos : 1	pos : 2	pos : 3		

Implementing a Concrete Syntax Tree

input tokens: `a + 2`

CST





JavaScript Tree Demo

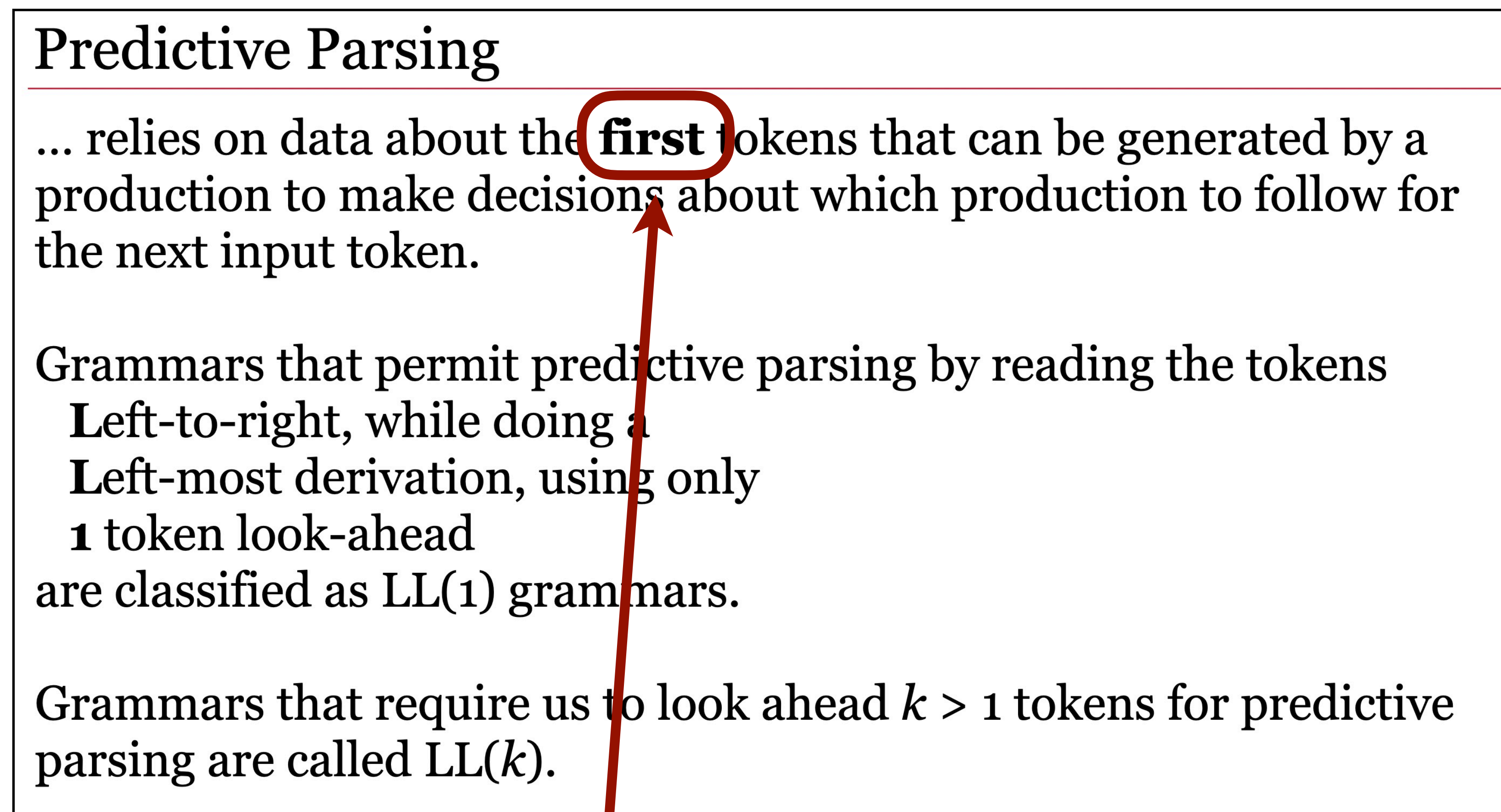
<Root>
-<Expr>
--<Term>
---<Factor>
----[a]
-<Op>
--[+]
-<Term>
--<Factor>
---[2]

Source Code

Back to Labouseur.com

Predictive Parsing

Remember this slide from a while back?
About what makes a grammar LL(1)?


Predictive Parsing

… relies on data about the **first** tokens that can be generated by a production to make decisions about which production to follow for the next input token.

Grammars that permit predictive parsing by reading the tokens
Left-to-right, while doing a
Left-most derivation, using only
1 token look-ahead
are classified as LL(1) grammars.

Grammars that require us to look ahead $k > 1$ tokens for predictive parsing are called LL(k).

See how the word "first" was in bold? That was not an accident. As it turns out, we are very interested in the set of symbols that can come first in any given production. We'll call this the **first set**.

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*?
I.e., What denotes the start of the production immediately **follow**ing *P*?

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1. <A> ::== <B> a
2. <B> ::== b
3.     ::== c
```

FIRST(A) = { } FOLLOW(A) = { }
FIRST(B) = { } FOLLOW(B) = { }

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1.  <A>  ::== <B> a
2.  <B>  ::== b
3.       ::== c
```

FIRST(A) = { } FOLLOW(A) = { }
FIRST(B) = { } FOLLOW(B) = { }

<A> begins with <B> so we need to know about that before we can do anything here.

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1.  <A> ::== <B> a
2.  <B> ::== b
3.      ::== c
```

FIRST(A) = { }
FIRST(B) = {`b`, `c`}

FOLLOW(A) = { }
FOLLOW(B) = { }

<B> can begin with `b` or `c`, so those go in FIRST of <B>.

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1.  <A>  ::== <B> a
2.  <B>  ::== b
3.        ::== c
```

FIRST(A) = {**b**, **c**}   FOLLOW(A) = { }
FIRST(B) = {**b**, **c**}   FOLLOW(B) = { }

Now we know FIRST(A) since it begins with <B> and we know FIRST(B). We add FIRST(B) to FIRST(A).

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1.  <A> ::== <B> a
2.  <B> ::== b
3.      ::== c
```

FIRST(A) = {**b**, **c**}   FOLLOW(A) = {**$**}
FIRST(B) = {**b**, **c**}   FOLLOW(B) = { }

What about follow sets? <A> is the start symbol. The start symbol is always followed by the end-of-program marker (and possibly other symbols).

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1. <A> ::== <B> a
2. <B> ::== b
3.     ::== c
```

FIRST(A) = {b, c}   FOLLOW(A) = {$}
FIRST(B) = {b, c}   FOLLOW(B) = {a}

What about follow sets? Looking at the right-hand side of the rewrite rules, we see that **a** can come immediately after <B>. We add FIRST(**a**) to FOLLOW(B).

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*?
I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1.  <A> ::== <B> a
2.  <B> ::== b
3.      ::== c
```

FIRST(A) = {b, c}   FOLLOW(A) = {$}
FIRST(B) = {b, c}   FOLLOW(B) = {a}

There are no more possibilities for FIRST and FOLLOW in this grammar so we're done. In grammars more interesting than this one we'll have to make more passes to get everything.

First and Follow Sets

First Set
Given a production, *P*, what terminal symbols indicate that *P* is just beginning? I.e., What denotes the start of *P*? What is the **first** of *P*?

Follow Set
What terminal symbols can we see coming immediately after *P*? I.e., What denotes the start of the production immediately **follow**ing *P*?

Consider the following grammar:

```
1. <A> ::== <B> a
2. <B> ::== b
3.     ::== c
```

FIRST(A) = {**b**, **c**}   FOLLOW(A) = {**$**}
FIRST(B) = {**b**, **c**}   FOLLOW(B) = {**a**}

We can also write it in table form.

?	FIRST	FOLLOW
<A>	{ b,c }	{ $ }
<B>	{ b,c }	{ a }

First and Follow Sets

First Sets

FIRST(*terminal symbol*) = *terminal symbol*

FIRST(*non-terminal*) = the union of the first sets of all the productions for that non-terminal

Algorithm for computing FIRST (α)

1. If α = **ε** (epsilon production) then α is nullable.
2. If α starts with a terminal symbol then FIRST (α) contains that terminal symbol.
3. If α starts with a non-terminal then add FIRST (that non-terminal) to FIRST (α) and…
   a. stop if that non-terminal does not contain any **ε** productions; or
   b. continue adding FIRST (non-terminals or terminals after that) until you reach one without any **ε** productions. I.e., when you add a nullable symbol to a first set you must "keep going" because that symbol could be skipped and we need to account for everything that might start α.

First and Follow Sets

Follow Sets

Algorithm for computing FOLLOW sets

1. Start by adding the end-of-program symbol to FOLLOW (*start symbol*).
2. For each production, *P*, and each instance of *P* in another production "followed by" *Y*, add FIRST(*Y*) to FOLLOW(*P*).

Consider the following grammar:

```
1. <A> ::== <B> a
2. <B> ::== b
3.      ::== c
```

FIRST(A) = {b, c}   FOLLOW(A) = {$}
FIRST(B) = {b, c}   FOLLOW(B) = {a}

3. For each production, *P*, and instance of *P* at the end of a production, *Z*, add FOLLOW(*Z*) to FOLLOW(*P*).

```
<Z> ::= a b <P>
```

Why? Because Z can end in a P, so we need to make sure that whatever can FOLLOW(*Z*) can FOLLOW(*P*).

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

	nullable?	FIRST	FOLLOW
<Z>		{ }	{ }
<Y>		{ }	{ }
<X>		{ }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST Pass #1

Sweep through and note the easy stuff (terminals) to get started.

	nullable?	FIRST	FOLLOW
<Z>		{ d }	{ }
<Y>	*yes*	{ c }	{ }
<X>		{ a }	{ }

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

FIRST Pass #2a

<Z> can start with an <X> so add FIRST(X) to FIRST(Z).

	nullable?	FIRST	FOLLOW
<Z>		{ d, a }	{ }
<Y>	*yes*	{ c }	{ }
<X>		{ a }	{ }

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

FIRST Pass #2b

<X> can start with a <Y> so add FIRST(Y) to FIRST(X). Also, note that <Y> is nullable, which means that <X> is too since nothing follows <Y> in production #5.

	nullable?	FIRST	FOLLOW
<Z>		{ d, a }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST Pass #3a

<Z> can start with an <X> so add FIRST(X) — which has changed since the last pass — to FIRST(Z).

	nullable?	FIRST	FOLLOW
<Z>		{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	a, c	{ }

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

FIRST Pass #3b

<X> is nullable so <Z> might start with a <Y> so add FIRST(Y) to FIRST(Z).
(No change, but we must consider it.)

	nullable?	FIRST	FOLLOW
<Z>		{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST Pass #3c

<Y> is also nullable so <Z> might just start with a <Z> so add FIRST(Z) to FIRST(Z). This does nothing but we must consider it.

	nullable?	FIRST	FOLLOW
<Z>		{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST Pass #3d

Even though <X> and <Y> are nullable <Z> is not because we'll still need a <Z> to end production #2, and it will contain **d**.

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST Pass #4

No change.

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST set complete
(And nullability too.)

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FOLLOW Pass #1a

Add the end-of-program marker to the follow set of the start symbol.

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ }

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

FOLLOW Pass #1b

<X> can be followed by <Y> so add FIRST(Y) to FOLLOW(X).

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ c }

First and Follow Sets

Consider the following grammar:

```
1.  <Z> ::== d
2.      ::== <X> <Y> <Z>
3.  <Y> ::== ε
4.      ::== c
5.  <X> ::== <Y>
6.      ::== a
```

?

FOLLOW Pass #1c

<Y> is nullable so consider that it might not be there and add FIRST(Z) to FOLLOW(X).

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FOLLOW Pass #1d

<Y> can be followed by <Z> so add FIRST(Z) to FOLLOW(Y).

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

?

FOLLOW Pass #1e

<Y> is nullable so consider that it might not be there and add FIRST(Z) to FOLLOW(X). (No change. But it's smart to consider it again.)

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FOLLOW Pass #1f

<X> ends in a <Y> so we have to make sure that whatever can follow an <X> can follow a <Y>. Add FOLLOW(X) to FOLLOW(Y). (No change. But we have to consider it.)

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1.  <Z> ::== d
2.      ::== <X> <Y> <Z>
3.  <Y> ::== ε
4.      ::== c
5.  <X> ::== <Y>
6.      ::== a
```

FOLLOW Pass #2a

<Z> ends in a <Z> so we have to make sure that whatever can follow an <Z> can follow a <Z>. (Huh?) Add FOLLOW(Z) to FOLLOW(Z). (No change. But we had to ponder it, for completeness if nothing else.)

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FOLLOW Pass #2b

Nothing changed.

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

FIRST and FOLLOW sets complete.
(And nullability too.)

Now we can make a Parse Table.

	nullable?	FIRST	FOLLOW
<Z>	*no*	{ d, a, c }	{ $ }
<Y>	*yes*	{ c }	{ d, a, c }
<X>	*yes*	{ a, c }	{ d, a, c }

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

To build a Parse Table cross-reference productions with terminals based on the elements of the productions' first and follow sets.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }			
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

<Z> is the start symbol. We need to fill in the parse table for all symbols in FIRST<Z>.

<Z> is not nullable so we do not need to look at its follow set.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }			
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

<Z> is the start symbol. We need to fill in the parse table for all symbols in FIRST<Z>.

```
<Z> ::== d
```

is production #1 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }			#1
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

<Z> is the start symbol.
We need to fill in the parse table for all symbols in FIRST<Z>.

```
<Z> ::== <X> <Y> <Z>
```

is production #2 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2		#1
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

<Z> is the start symbol.
We need to fill in the parse table for all symbols in FIRST<Z>.

```
<Z> ::== <X> <Y> <Z>
```

is production #2 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

<Z> is the start symbol.
We need to fill in the parse table for all symbols in FIRST<Z>.

```
<Z> ::== <X> <Y> <Z>
```

is production #2 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z> ::== d
2.          ::== <X> <Y> <Z>
3.  <Y> ::== ε
4.          ::== c
5.  <X> ::== <Y>
6.          ::== a
```

Parse Table

Next we need to fill in the table for all symbols in FIRST<Y>.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }			
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

Next we need to fill in the table for all symbols in FIRST<Y>.

```
<Y> ::== c
```

is production #4 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }		#4	
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Since <Y> is nullable it may be skipped via production #3 so we need to account for all symbols in FOLLOW<Y> and cross-reference them to production #3.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }		#4	
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.        ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.        ::== c
5.  <X>  ::== <Y>
6.        ::== a
```

Parse Table

Since <Y> is nullable it may be skipped via production #3 so we need to account for all symbols in FOLLOW<Y> and cross-reference them to production #3.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }		#4	#3
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Since <Y> is nullable it may be skipped via production #3 so we need to account for all symbols in FOLLOW<Y> and cross-reference them to production #3.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#4	#3
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Since <Y> is nullable it may be skipped via production #3 so we need to account for all symbols in FOLLOW<Y> and cross-reference them to production #3.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.            ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.            ::== c
5.  <X>  ::== <Y>
6.            ::== a
```

Parse Table

Finally, we need to fill in the table for all symbols in FIRST<X>.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }			

First and Follow Sets

Consider the following grammar:

```
1.  <Z> ::== d
2.      ::== <X> <Y> <Z>
3.  <Y> ::== ε
4.      ::== c
5.  <X> ::== <Y>
6.      ::== a
```

Parse Table

Finally, we need to fill in the table for all symbols in FIRST<X>.

```
<X> ::== a
```

is production #6 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#6		

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Finally, we need to fill in the table for all symbols in FIRST<X>.

```
<X>  ::== <Y>
```

is production #5 so put it in the parse table.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#6	#5	

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Since <X> is nullable it may be skipped via production #5 so we need to account for all symbols in FOLLOW<X> and cross-reference them to production #5.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#6	#5	

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

Since <X> is nullable it may be skipped via production #5 so we need to account for all symbols in FOLLOW<X> and cross-reference them to production #5.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#6	#5	#5

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.       ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.       ::== c
5.  <X>  ::== <Y>
6.       ::== a
```

Parse Table

Since <X> is nullable it may be skipped via production #5 so we need to account for all symbols in FOLLOW<X> and cross-reference them to production #5.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#5, #6	#5	#5

First and Follow Sets

Consider the following grammar:

```
1.  <Z>  ::== d
2.        ::== <X> <Y> <Z>
3.  <Y>  ::== ε
4.        ::== c
5.  <X>  ::== <Y>
6.        ::== a
```

Parse Table

Since <X> is nullable it may be skipped via production #5 so we need to account for all symbols in FOLLOW<X> and cross-reference them to production #5.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#5, #6	#5	#5

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

The Parse Table is now complete.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	#2	#2	#1, #2
<Y>	*yes*	{ c }	{ d, a, c }	#3	#3, #4	#3
<X>	*yes*	{ a, c }	{ d, a, c }	#5, #6	#5	#5

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

Parse Table

Sometimes the Parse Table is written with the actual productions instead of their numbers.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	`<Z>::== <X> <Y> <Z>`	`<Z>::== <X> <Y> <Z>`	`<Z> ::== d` `<Z>::== <X> <Y> <Z>`
<Y>	*yes*	{ c }	{ d, a, c }	`<Y> ::== ε`	`<Y> ::== ε` `<Y> ::== c`	`<Y> ::== ε`
<X>	*yes*	{ a, c }	{ d, a, c }	`<X> ::== <Y>` `<X> ::== a`	`<X> ::== <Y>`	`<X> ::== <Y>`

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

The Parse Table tells us whether or not the grammar is LL(1).

A grammar is LL(1) if all entries in the parse table contain only one production.

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	<Z>::== <X> <Y> <Z>	<Z>::== <X> <Y> <Z>	<Z> ::== d <Z>::== <X> <Y> <Z>
<Y>	*yes*	{ c }	{ d, a, c }	<Y> ::== ε	<Y> ::== ε <Y> ::== c	<Y> ::== ε
<X>	*yes*	{ a, c }	{ d, a, c }	<X> ::== <Y> <X> ::== a	<X> ::== <Y>	<X> ::== <Y>

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

The Parse Table tells us whether or not the grammar is LL(1).

A grammar is LL(1) if all entries in the parse table contain only one production.

This grammar is **not** LL(1).

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	<Z>::== <X> <Y> <Z>	<Z>::== <X> <Y> <Z>	<Z> ::== d <Z>::== <X> <Y> <Z>
<Y>	*yes*	{ c }	{ d, a, c }	<Y> ::== ε	<Y> ::== ε <Y> ::== c	<Y> ::== ε
<X>	*yes*	{ a, c }	{ d, a, c }	<X> ::== <Y> <X> ::== a	<X> ::== <Y>	<X> ::== <Y>

First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

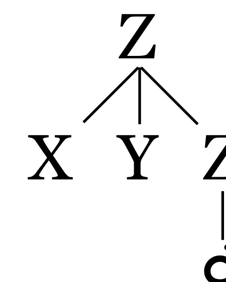
The Parse Table tells us whether or not the grammar is LL(1).

A grammar is LL(1) if all entries in the parse table contain only one production.

This grammar is **not** LL(1).

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	`<Z>::== <X> <Y> <Z>`	`<Z>::== <X> <Y> <Z>`	`<Z> ::== d` `<Z>::== <X> <Y> <Z>`
<Y>	*yes*	{ c }	{ d, a, c }	`<Y> ::== ε`	`<Y> ::== ε` `<Y> ::== c`	`<Y> ::== ε`
<X>	*yes*	{ a, c }	{ d, a, c }	`<X> ::== <Y>` `<X> ::== a`	`<X> ::== <Y>`	`<X> ::== <Y>`

Consider just the parse table entries for <Z> with input **d**. There are two possible parse trees. We'd need backtracking to parse this because it's **vague**. It's not LL(1).



First and Follow Sets

Consider the following grammar:

```
1. <Z> ::== d
2.     ::== <X> <Y> <Z>
3. <Y> ::== ε
4.     ::== c
5. <X> ::== <Y>
6.     ::== a
```

The Parse Table tells us whether or not the grammar is LL(1).

A grammar is LL(1) if all entries in the parse table contain only one production.

This grammar is **not** LL(1).

	nullable?	FIRST	FOLLOW	a	c	d
<Z>	*no*	{ d, a, c }	{ $ }	<Z>::== <X> <Y> <Z>	<Z>::== <X> <Y> <Z>	<Z> ::== d <Z>::== <X> <Y> <Z>
<Y>	*yes*	{ c }	{ d, a, c }	<Y>		
<X>	*yes*	{ a, c }	{ d, a, c }	<X> : <X>		

Consider just the parse table entries for <Z> with input `d`. There are two possible parse trees. We'd need backtracking to parse this because it's **vague**. It's not LL(1).

Many books and references use the term **ambiguous** instead of **vague**. "Ambiguous" in the context of LL(1)-*ness* and Parse Tables is separate and apart from its use in describing grammars that can produce different CSTs for the same input. We must be careful with our terminology here. That's why I like to call non-LL(1) grammars *vague* instead of *ambiguous*.

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 }	{ ? }												

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

```
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.

Error Recovery

So we're parsing and detect an error. Report it. Then what?

Q: How do we recover from a parse error and continue parsing to detect additional actual errors without producing meaningless errors after the initial error has thrown us out of sync with the grammar? (Spurious error messages tend to be ignored, which leads programmers to ignore the meaningful errors too. So it's important that we report only actual errors.)

Error Recovery

So we're parsing and detect an error. Report it. Then what?

Q: How do we recover from a parse error and continue parsing to detect additional actual errors without producing meaningless errors after the initial error has thrown us out of sync with the grammar? (Spurious error messages tend to be ignored, which leads programmers to ignore the meaningful errors too. So it's important that we report only actual errors.)

A: Look to the follow set. When we detect and report an error parsing production *Px* (in *parsePx*() probably) we can begin skipping tokens until we find one in FOLLOW(*Px*). Then we're back in sync with the grammar and can restart parsing and reporting errors again.