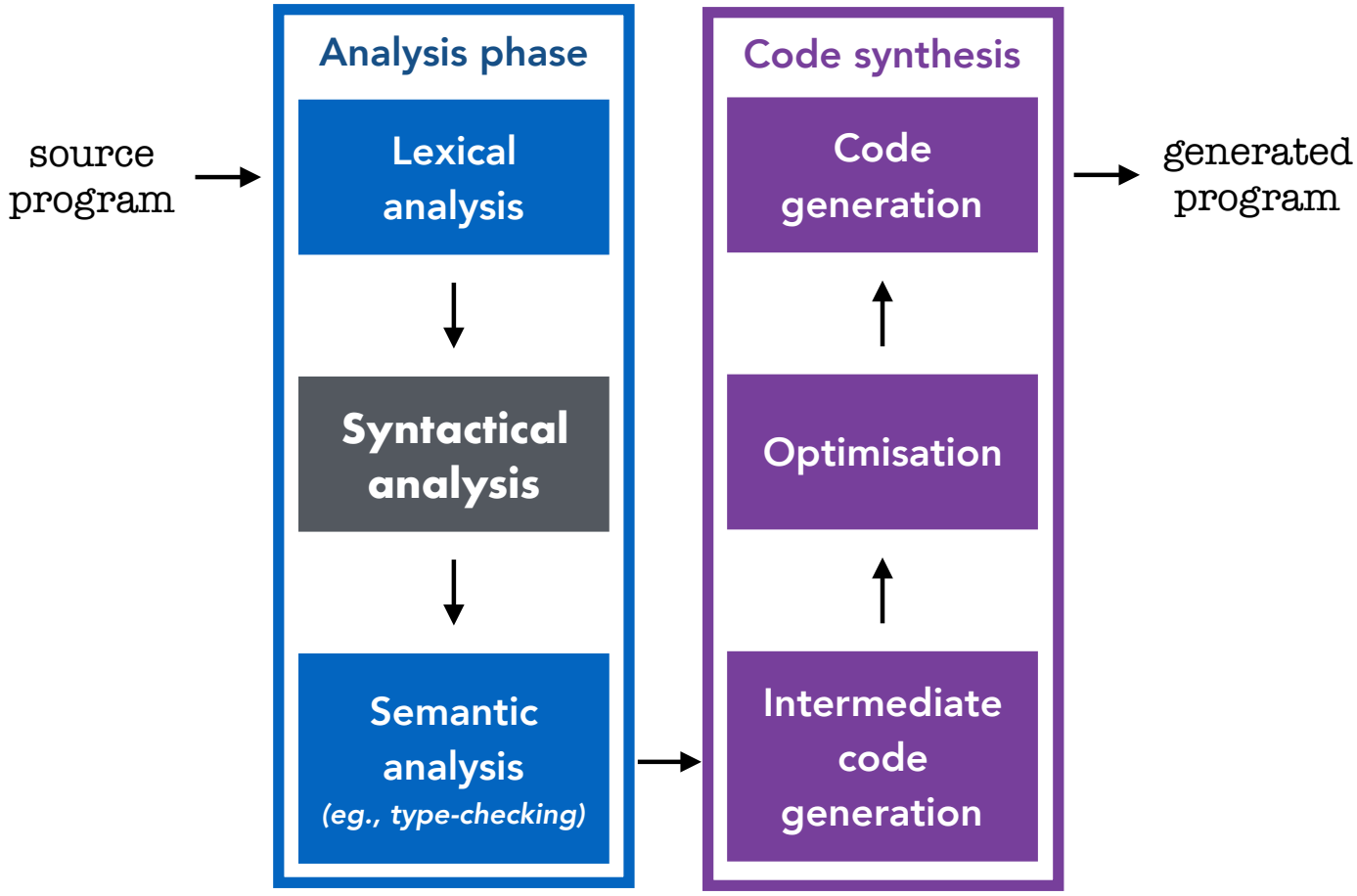
Compilers and computer architecture

6. From strings to ASTs: parse trees

Ian Mackie

Autumn 2022

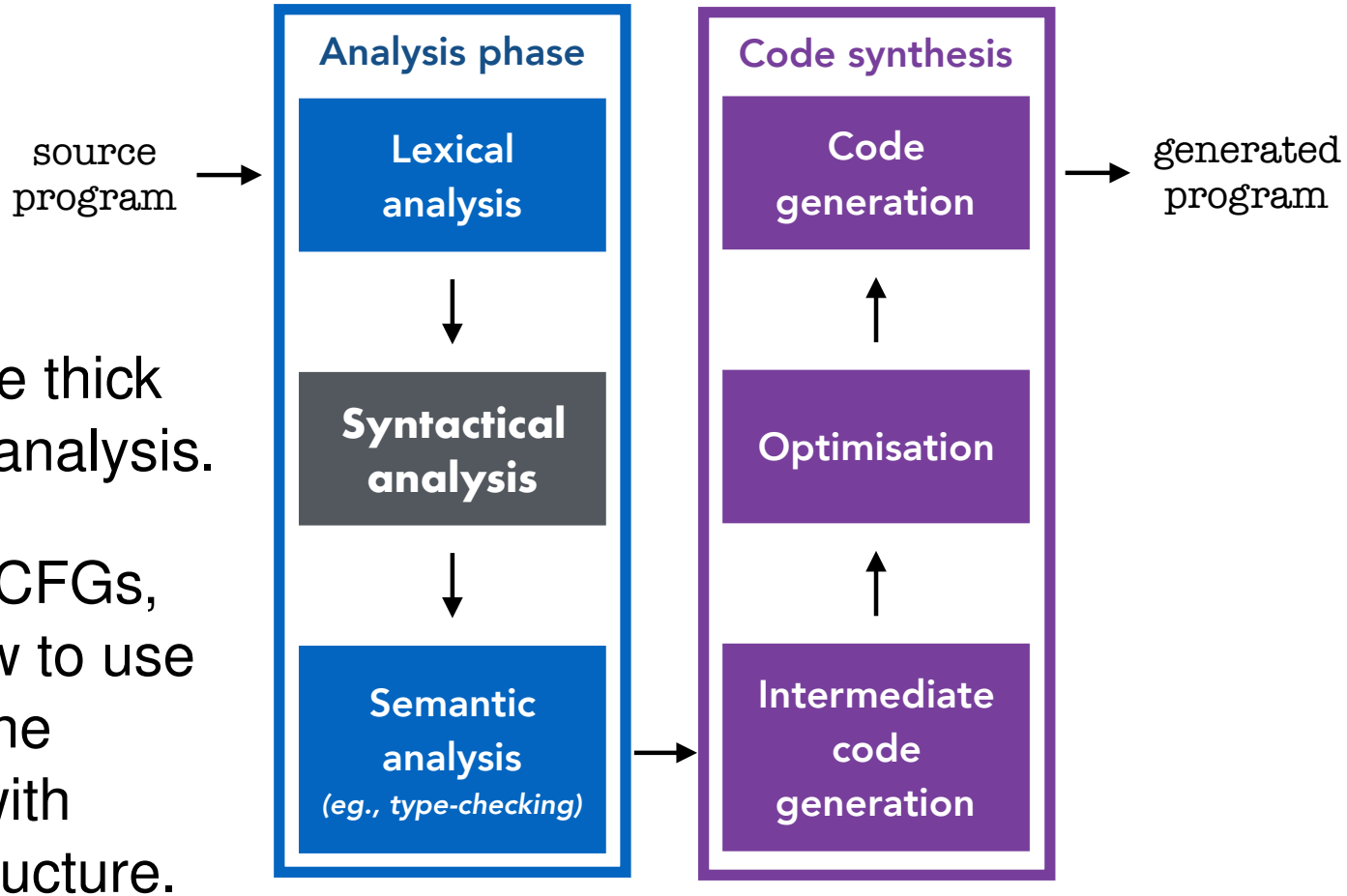
Where we are


source program
Analysis phase
Lexical analysis
Syntactical analysis
Semantic analysis
(eg., type-checking)
Code synthesis
Intermediate code generation
Optimisation
Code generation
generated program

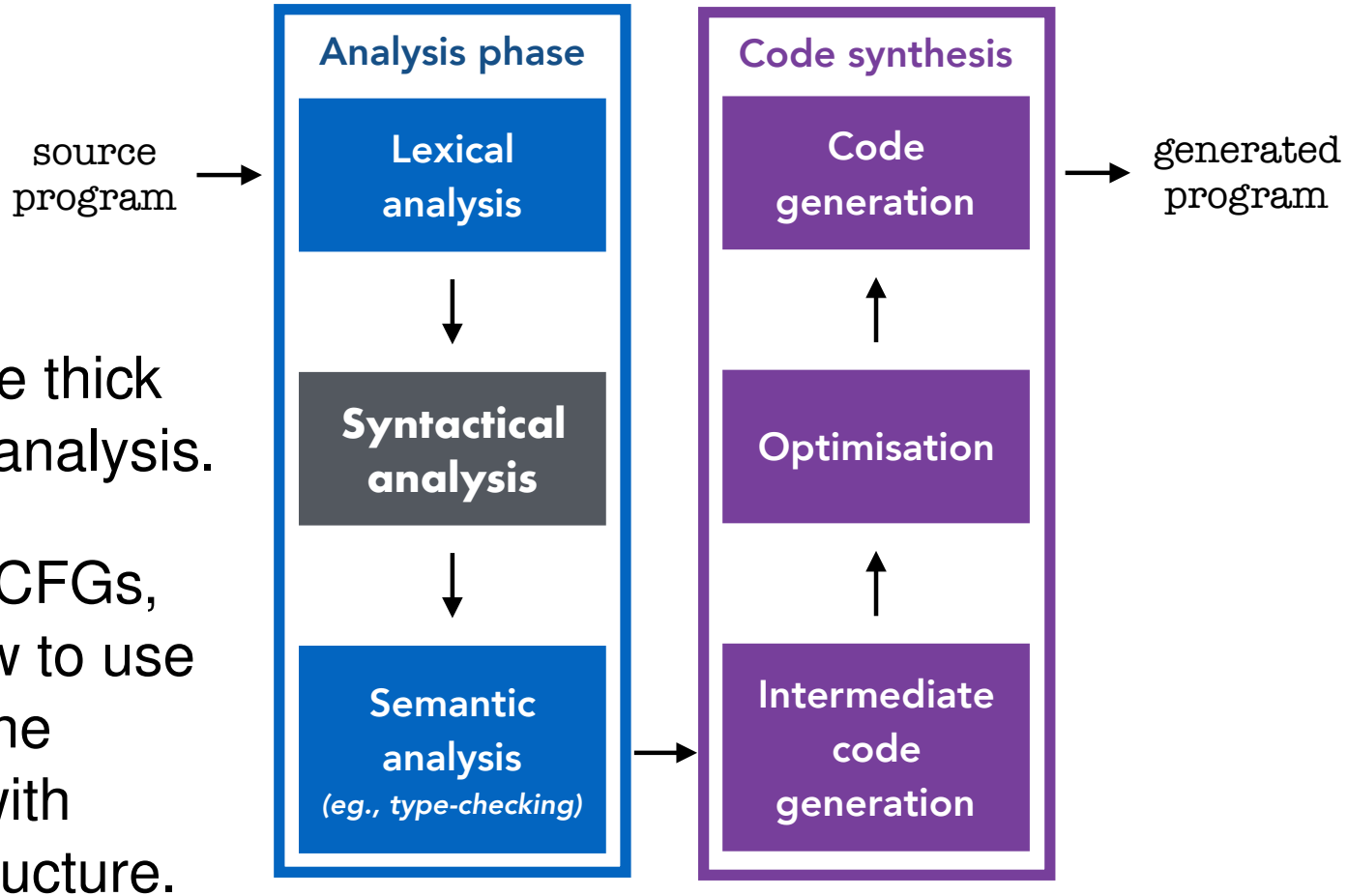
Where we are

We are in the thick of syntactic analysis.

We defined CFGs, and saw how to use them to define languages with recursive structure.



Where we are



We are in the thick of syntactic analysis.

We defined CFGs, and saw how to use them to define languages with recursive structure.

But how do we use CFGs to:

- represent a programming language?
- extract the structure of a program (*i.e.*, construct an AST)?

From strings to ASTs

We want to take a program given as a string and:

- Check if it's syntactically correct, *e.g.*:
  - Is every open bracket later closed?
  - Is it a sequence of valid declarations / statements?

From strings to ASTs

We want to take a program given as a string and:

- Check if it's syntactically correct, *e.g.*:
  - Is every open bracket later closed?
  - Is it a sequence of valid declarations / statements?
- Produce an AST, to facilitate efficient **code generation**.

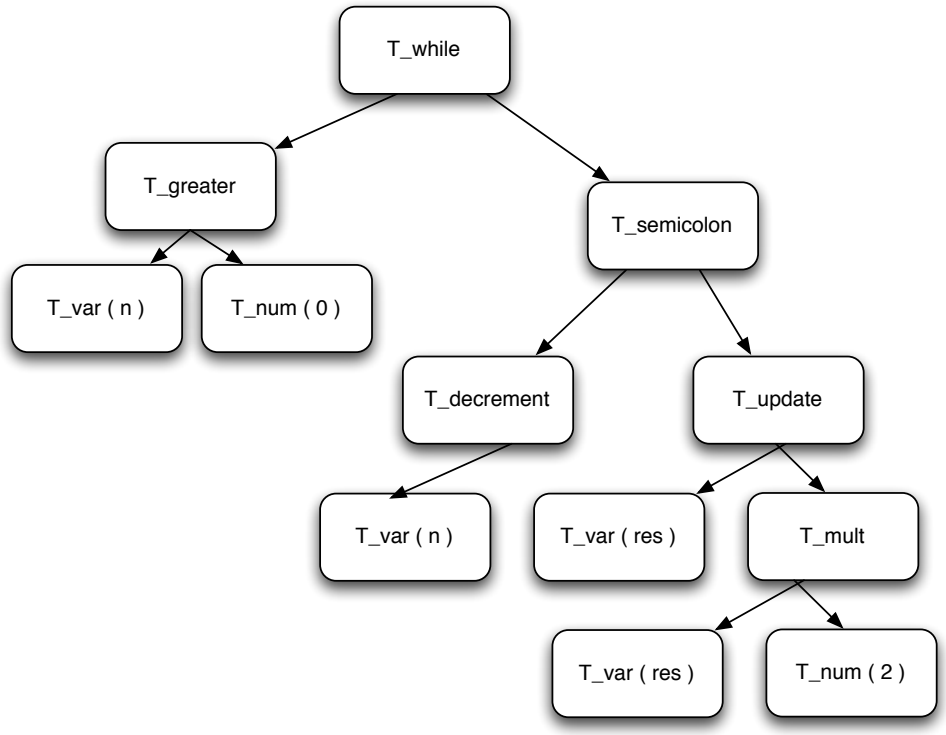
From strings to ASTs

We want to take a program given as a string and:

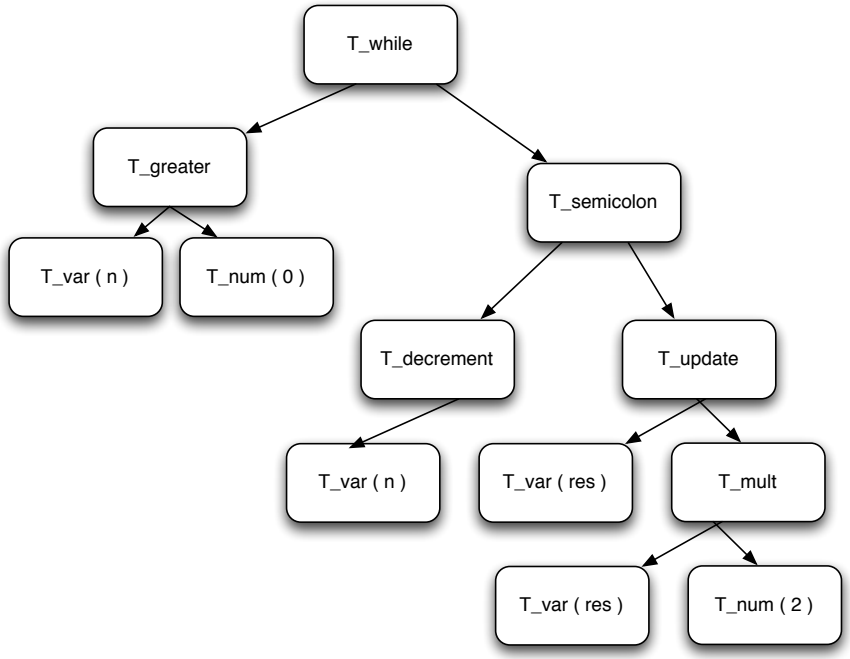
- Check if it's syntactically correct, *e.g.*:
  - Is every open bracket later closed?
  - Is it a sequence of valid declarations / statements?
- Produce an AST, to facilitate efficient **code generation**.

Recall, we want to go from the representation on the left, to that on the right:

```
while (n > 0) {
  n--;
  res *= 2;
}
```

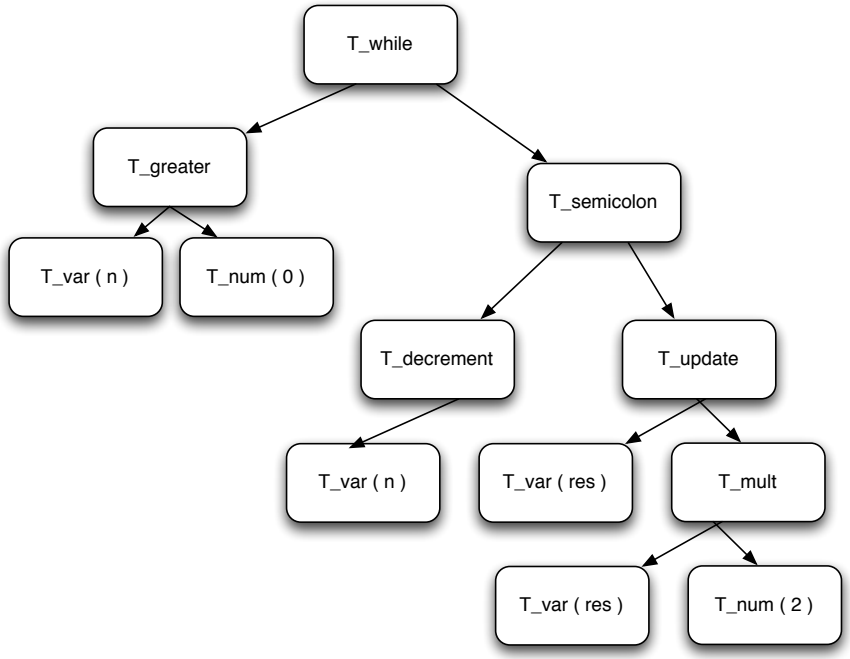


What is an AST
— ?



Question. We keep going on about these ASTs.
But what are they??

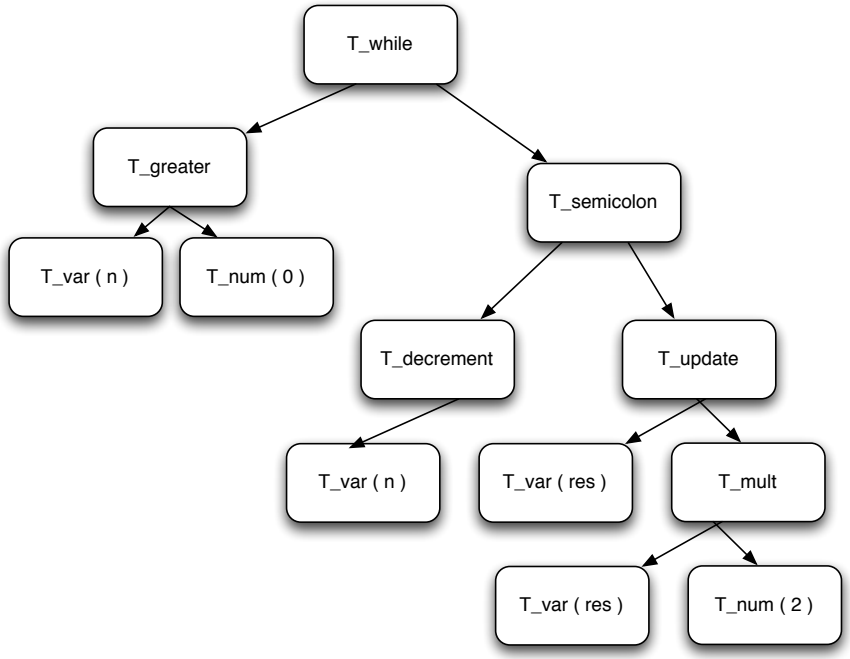
What is an AST
— ?



Question. We keep going on about these ASTs.
But what are they??

Answer. A data structure, which represents
the **derivation** of an input from a CFG.

What is an AST
— ?



Question. We keep going on about these ASTs.
But what are they??

Answer. A data structure, which represents
the **derivation** of an input from a CFG.

To see this more easily, we introduce: **parse trees**.

Derivations and parse trees

Recall that a derivation of a string $t \in A^*$
for a CFG (A, V, S, R), is a sequence

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

where:

- $S \in V$ is the initial variable (drawn from the set of variables V)
- the strings s_i may have symbols from both A and V in them

Derivations and parse trees

Recall that a derivation of a string $t \in A^*$
for a CFG (A, V, S, R), is a sequence

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

where:

- $S \in V$ is the initial variable (drawn from the set of variables V)
- the strings s_i may have symbols from both A and V in them
- each step $s_i \Rightarrow s_{i+1}$ is a "one-step reduction"
  - a substitution inside the string s_i , of a single variable $X \in V$, by some string — using the reductions given by R

Derivations and parse trees

Recall that a derivation of a string $t \in A^*$
for a CFG (A, V, S, R), is a sequence

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

where:

- $S \in V$ is the initial variable (drawn from the set of variables V)
- the strings s_i may have symbols from both A and V in them
- each step $s_i \Rightarrow s_{i+1}$ is a "one-step reduction"
  - a substitution inside the string s_i , of a single variable $X \in V$, by some string — using the reductions given by R

We can represent these derivations by **parse trees**,
which describe the pattern of substitutions
from variables to strings.

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

- The start symbol S is the root of the tree.

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

- The start symbol S is the root of the tree.
- The symbols of the string $s_1 = r_1 r_2 \ldots r_k$ (listed from left to right) are the children of S in the tree.

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

- The start symbol S is the root of the tree.
- The symbols of the string $s_1 = r_1 r_2 \ldots r_k$ (listed from left to right) are the children of S in the tree.
- For each reduction $X \to u_1 u_2 \ldots u_m$ in the derivation, we add nodes labeled $u_1, \ldots, u_m$ (listed from left to right) as children of X.

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

- The start symbol S is the root of the tree.
- The symbols of the string $s_1 = r_1 r_2 \ldots r_k$ (listed from left to right) are the children of S in the tree.
- For each reduction $X \to u_1 u_2 \ldots u_m$ in the derivation, we add nodes labeled $u_1, \ldots, u_m$ (listed from left to right) as children of X.
  - (or a node labeled with ε for a reduction $X \to \varepsilon$)

The structure of a parse tree

A parse tree is constructed for a given derivation

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow t$$

- The start symbol S is the root of the tree.
- The symbols of the string $s_1 = r_1 r_2 \ldots r_k$ (listed from left to right) are the children of S in the tree.
- For each reduction $X \to u_1 u_2 \ldots u_m$ in the derivation, we add nodes labeled $u_1, \ldots, u_m$ (listed from left to right) as children of X.
  - (or a node labeled with ε for a reduction $X \to \varepsilon$)

Then: nodes in the tree are labeled elements of $A \cup V \cup \{\varepsilon\}$

— and an in-order traversal of the *leaves* (removing any ε symbols) gives the string t

Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR $\rightarrow$ NUM | EXPR **+** EXPR | EXPR **–** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM $\rightarrow$ *(a number)*

Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR $\rightarrow$ NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM $\rightarrow$ *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR $\rightarrow$ NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM $\rightarrow$ *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

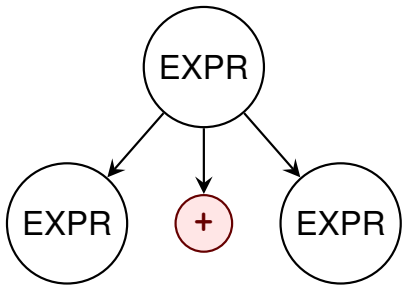
EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

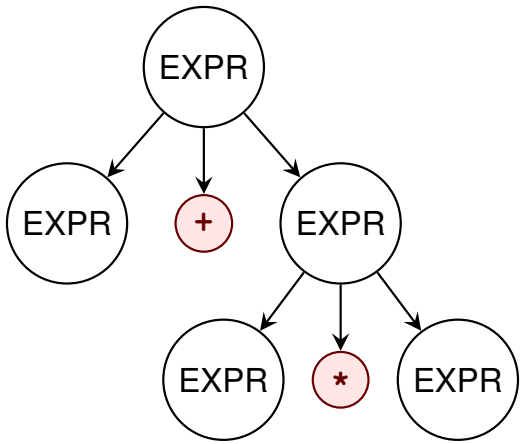
NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

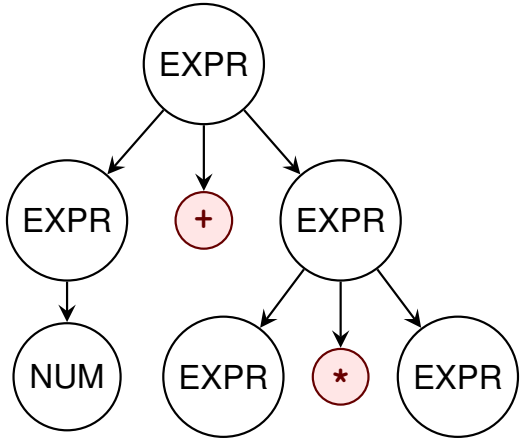
This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR

⇒ NUM `+` EXPR `*` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

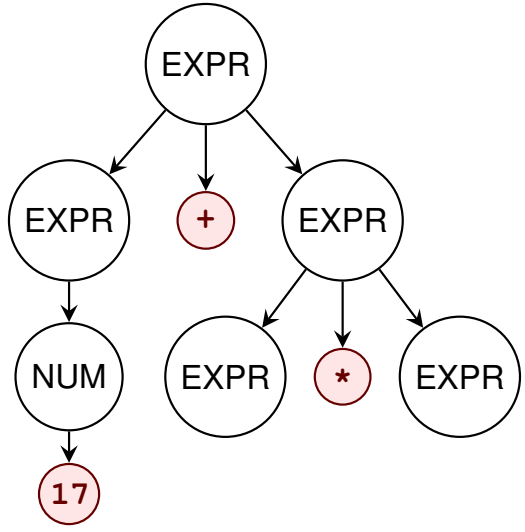
EXPR

⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR

⇒ NUM `+` EXPR `*` EXPR

⇒ `17 +` EXPR `*` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

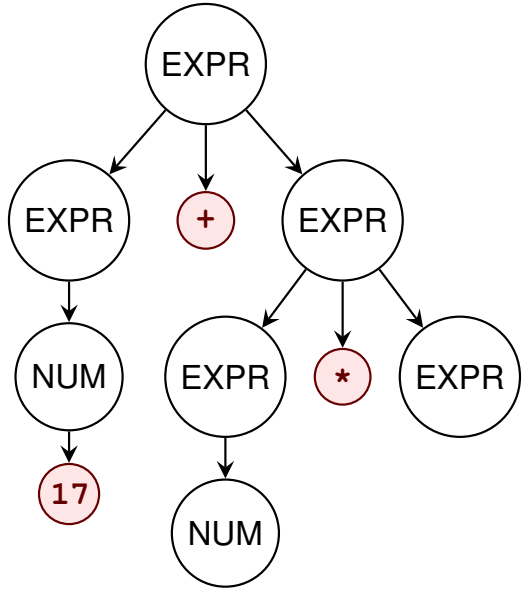
⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR

⇒ NUM `+` EXPR `*` EXPR

⇒ `17` `+` EXPR `*` EXPR

⇒ `17` `+` NUM `*` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR

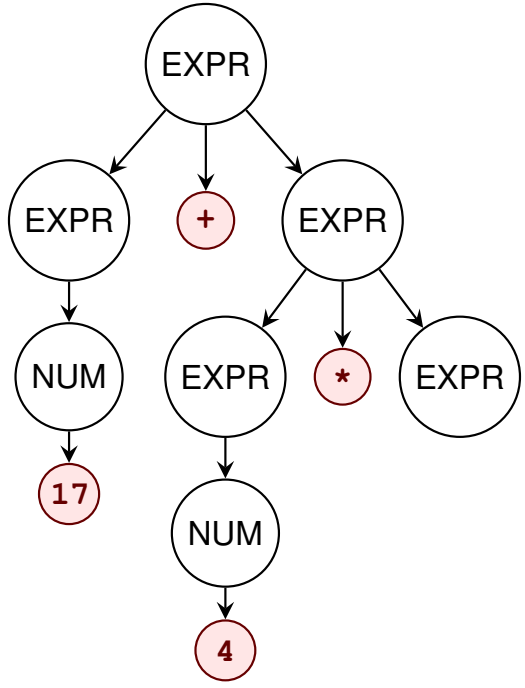
⇒ EXPR `+` EXPR `*` EXPR

⇒ NUM `+` EXPR `*` EXPR

⇒ `17 +` EXPR `*` EXPR

⇒ `17 +` NUM `*` EXPR

⇒ `17 + 4 *` EXPR



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR

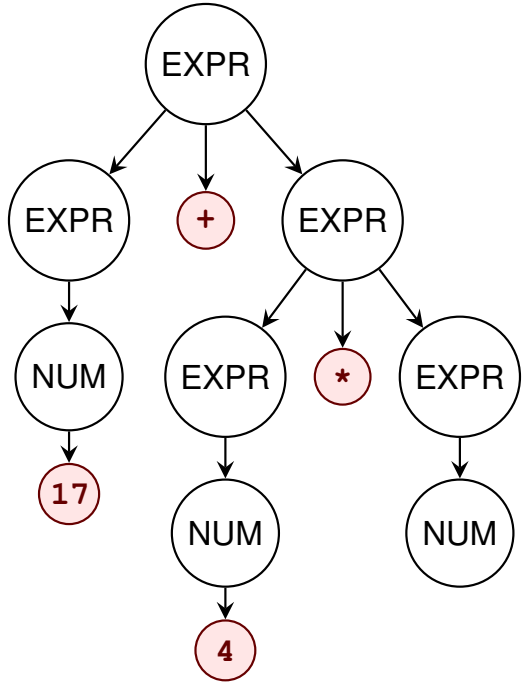
⇒ NUM `+` EXPR `*` EXPR

⇒ `17 +` EXPR `*` EXPR

⇒ `17 +` NUM `*` EXPR

⇒ `17 + 4 *` EXPR

⇒ `17 + 4 *` NUM



Example

A parse tree for simple expressions

A simple grammar for expressions over integers:

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

This language includes `17 + 4 * 3`, with *e.g.* the derivation:

EXPR

⇒ EXPR `+` EXPR

⇒ EXPR `+` EXPR `*` EXPR

⇒ NUM `+` EXPR `*` EXPR

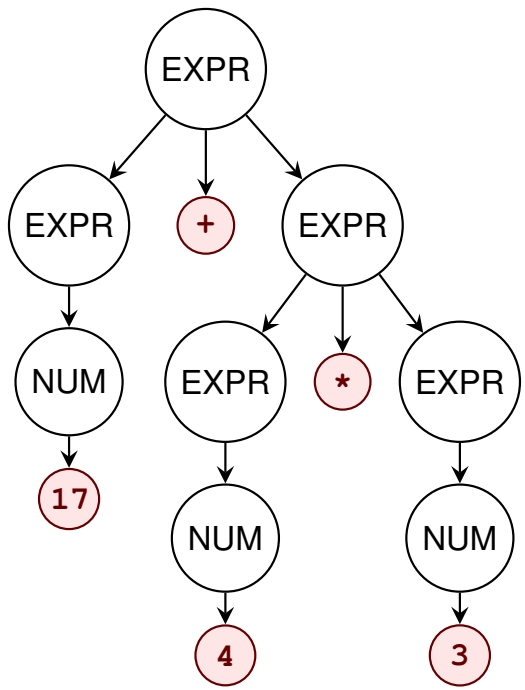
⇒ `17 +` EXPR `*` EXPR

⇒ `17 +` NUM `*` EXPR

⇒ `17 + 4 *` EXPR

⇒ `17 + 4 *` NUM

⇒ `17 + 4 * 3`



Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Derivation of “ **`if 2 < 3 print ("yes") else print ("no")`** ”:

Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Derivation of “ **`if 2 < 3 print ("yes") else print ("no")`** ”:

$$
\begin{aligned}
&\text{STMT} \\
&\Rightarrow^{*} \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT}
\end{aligned}
$$

Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Derivation of “ **`if 2 < 3 print ("yes") else print ("no")`** ”:

$$
\begin{aligned}
&\text{STMT} \\
&\Rightarrow^* \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
&\Rightarrow^* \texttt{if}\ \text{EXPR}\ \texttt{<}\ \text{EXPR}\ \texttt{then}\ \text{PROC}\ \texttt{else}\ \text{PROC}
\end{aligned}
$$

Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Derivation of “ **`if 2 < 3 print ("yes") else print ("no")`** ”:

$$
\begin{aligned}
&\text{STMT} \\
\Rightarrow^* &\ \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\Rightarrow^* &\ \texttt{if}\ \text{EXPR}\ \texttt{<}\ \text{EXPR}\ \texttt{then}\ \text{PROC}\ \texttt{else}\ \text{PROC} \\
\Rightarrow^* &\ \texttt{if}\ \text{NUM}\ \texttt{<}\ \text{NUM}\ \texttt{then}\ \text{IDEN}\ \texttt{(}\ \text{STRING}\ \texttt{)}\ \texttt{else}\ \text{IDEN}\ \texttt{(}\ \text{STRING}\ \texttt{)}
\end{aligned}
$$

Example

A derivation for an if statement

A simple grammar for part of a language with an **`if`** statement:

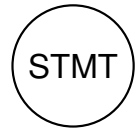
$$
\begin{aligned}
\text{STMT} &\to \text{IFSTMT} \mid \text{PROC} \mid \cdots \\
\text{IFSTMT} &\to \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\text{PROC} &\to \text{IDEN}\ \texttt{(}\ \texttt{)} \mid \text{IDEN}\ \texttt{(}\ \text{ARGS}\ \texttt{)} \\
\text{ARGS} &\to \text{EXPR} \mid \text{EXPR}\ \texttt{,}\ \text{ARGS} \\
\text{EXPR} &\to \text{IDEN} \mid \text{NUM} \mid \text{STRING} \mid \text{EXPR}\ \texttt{<}\ \text{EXPR} \mid \cdots \\
\text{IDEN} &\to \textit{(an identifier)} \qquad \text{NUM} \to \textit{(a number)} \qquad \text{STRING} \to \textit{(a string)}
\end{aligned}
$$

Derivation of “ **`if 2 < 3 print ("yes") else print ("no")`** ”:

$$
\begin{aligned}
&\text{STMT} \\
\Rightarrow^* &\ \texttt{if}\ \text{EXPR}\ \texttt{then}\ \text{STMT}\ \texttt{else}\ \text{STMT} \\
\Rightarrow^* &\ \texttt{if}\ \text{EXPR}\ \texttt{<}\ \text{EXPR}\ \texttt{then}\ \text{PROC}\ \texttt{else}\ \text{PROC} \\
\Rightarrow^* &\ \texttt{if}\ \text{NUM}\ \texttt{<}\ \text{NUM}\ \texttt{then}\ \text{IDEN}\ \texttt{(}\ \text{STRING}\ \texttt{)}\ \texttt{else}\ \text{IDEN}\ \texttt{(}\ \text{STRING}\ \texttt{)} \\
\Rightarrow^* &\ \texttt{if 2 < 3 then print ("yes") else print ("no")}
\end{aligned}
$$

Example

A parse tree for an if statement

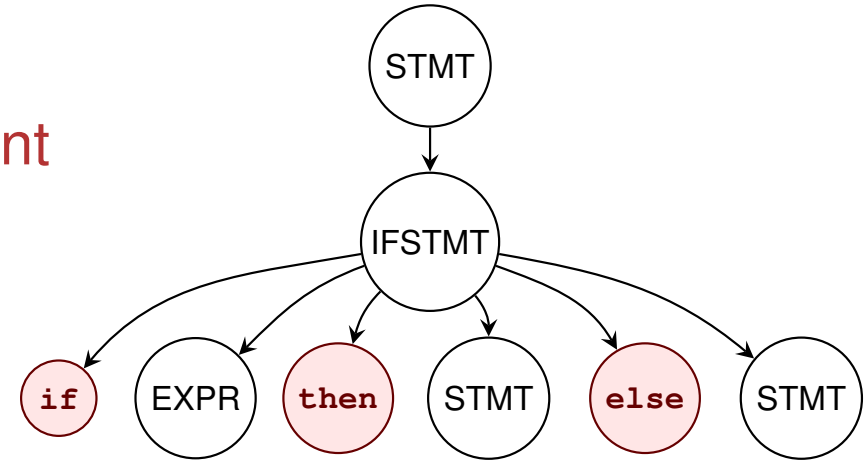


Generated string:

STMT

Example

A parse tree for an if statement

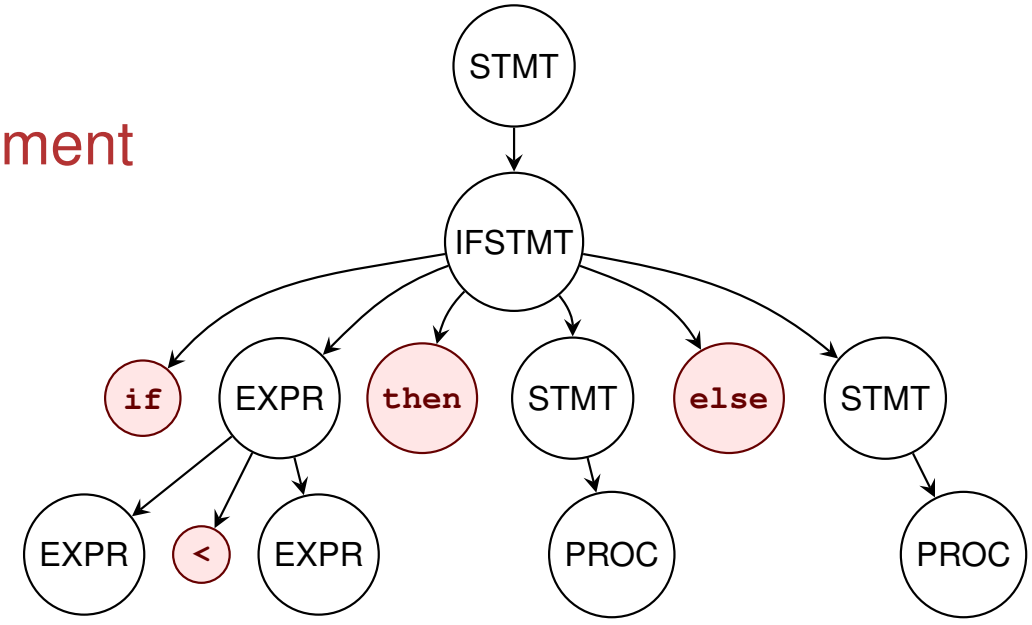


Generated string:

`if` EXPR **`then`** STMT **`else`** STMT

Example

A parse tree for an if statement

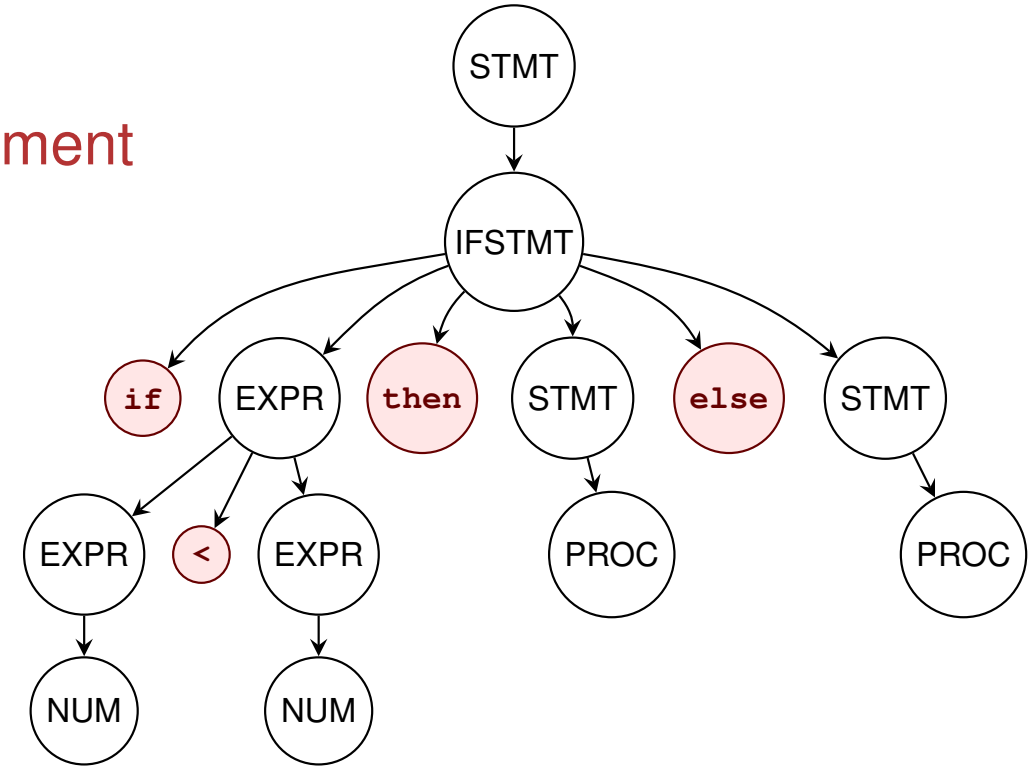


Generated string:

`if` EXPR **`<`** EXPR **`then`** PROC **`else`** PROC

Example

A parse tree for an if statement

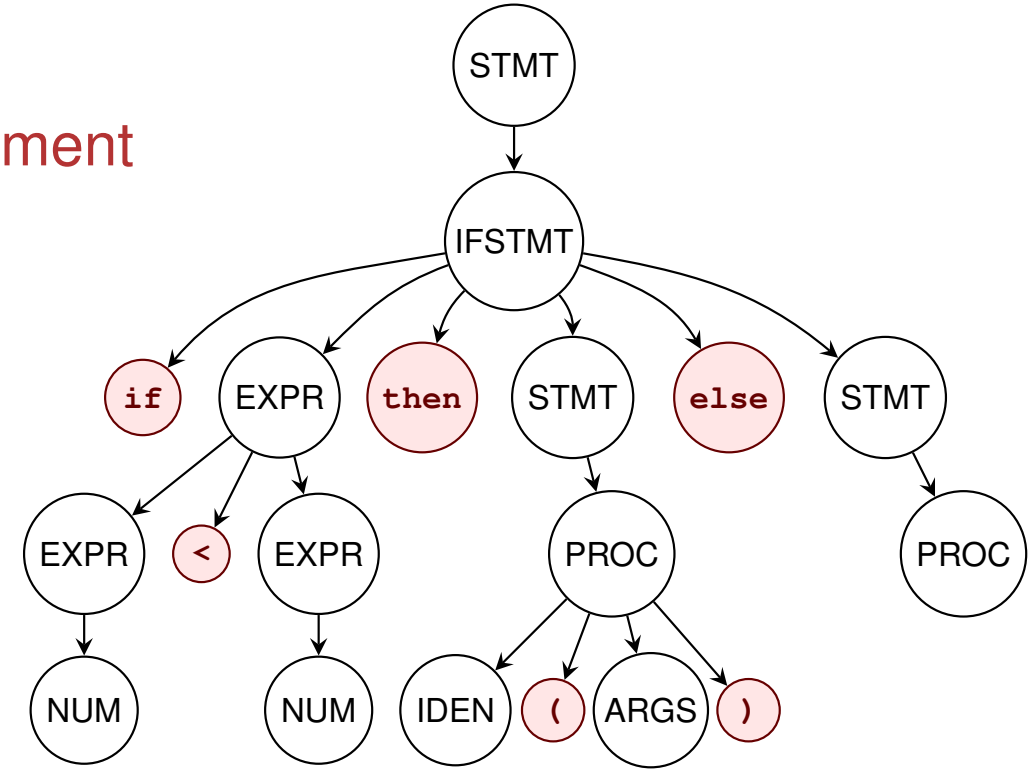


Generated string:

`if` NUM **`<`** NUM **`then`** PROC **`else`** PROC

Example

A parse tree for an if statement

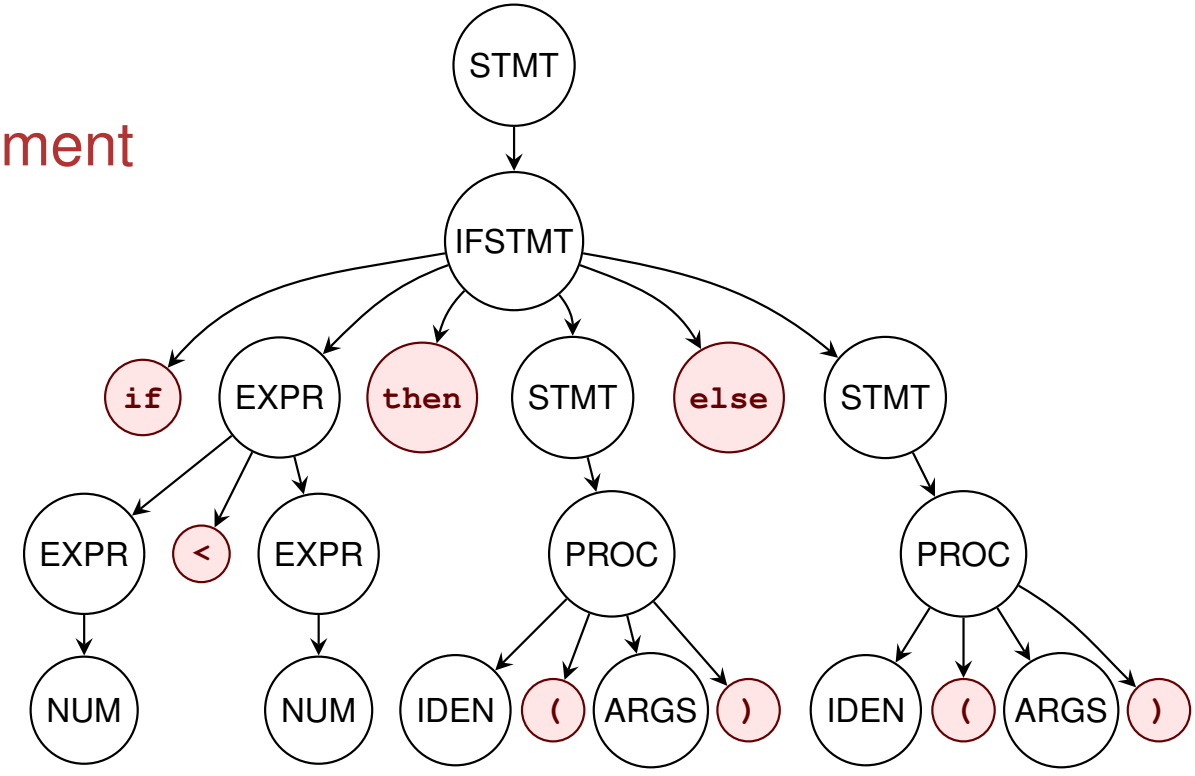


Generated string:

`if` NUM **`<`** NUM **`then`** IDEN **`(`** ARGS **`)`** **`else`** PROC

Example

A parse tree for an if statement

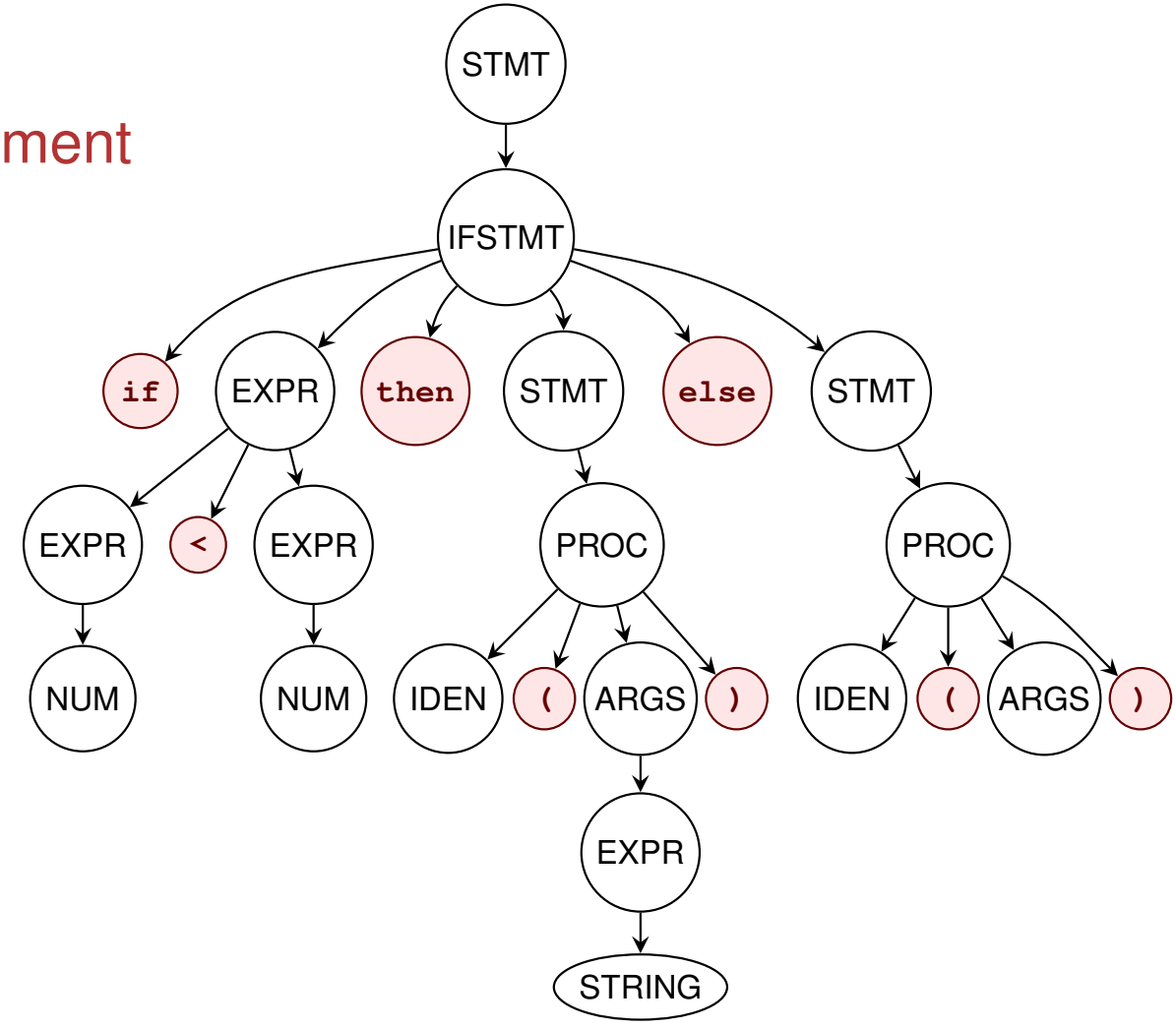


Generated string:

`if` NUM **`<`** NUM **`then`** IDEN **`(`** ARGS **`)`** **`else`** IDEN **`(`** ARGS **`)`**

Example

A parse tree for an if statement

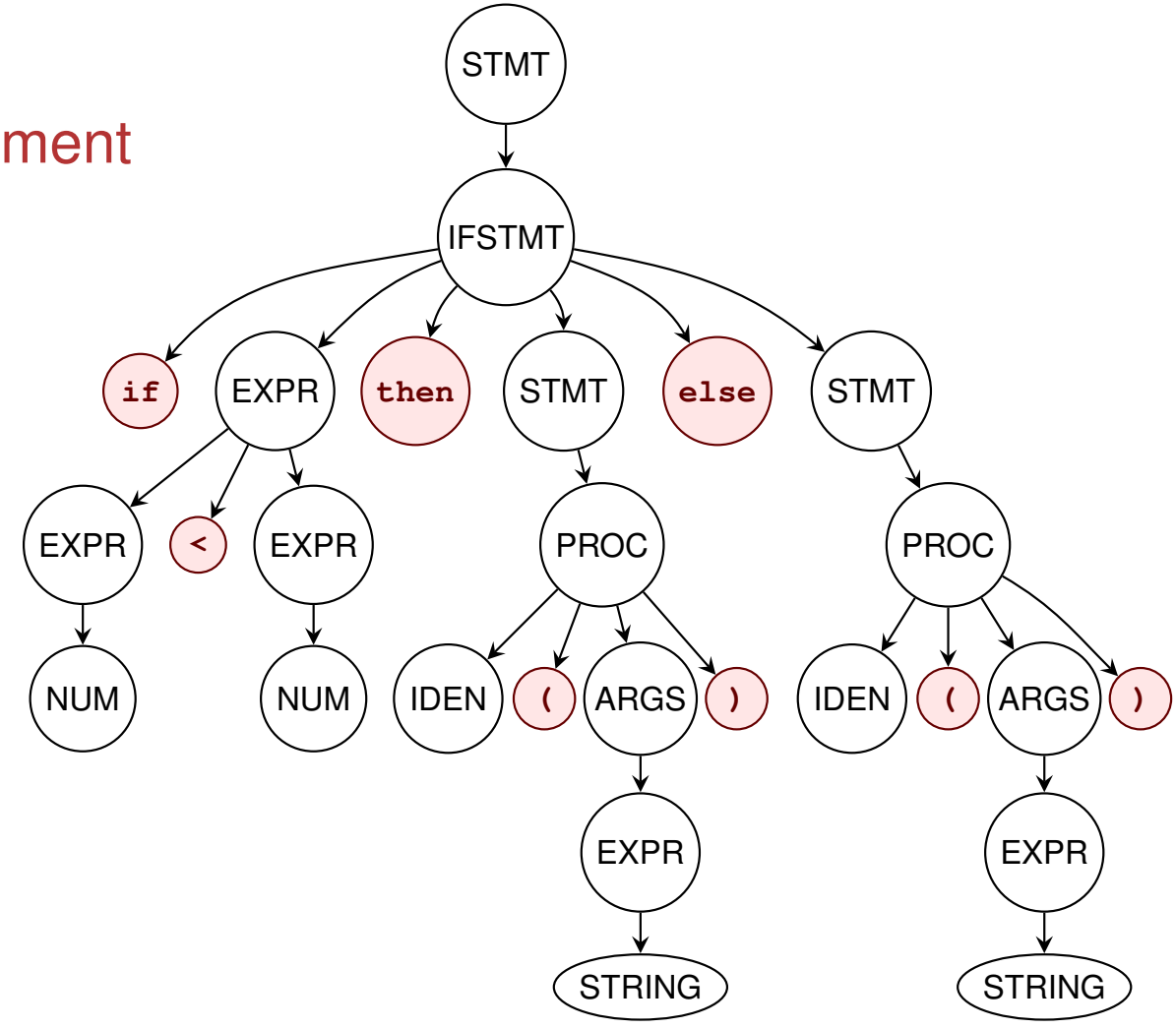


Generated string:

`if` NUM **`<`** NUM **`then`** IDEN **`(`** STRING **`)`** **`else`** IDEN **`(`** ARGS **`)`**

Example

A parse tree for an if statement

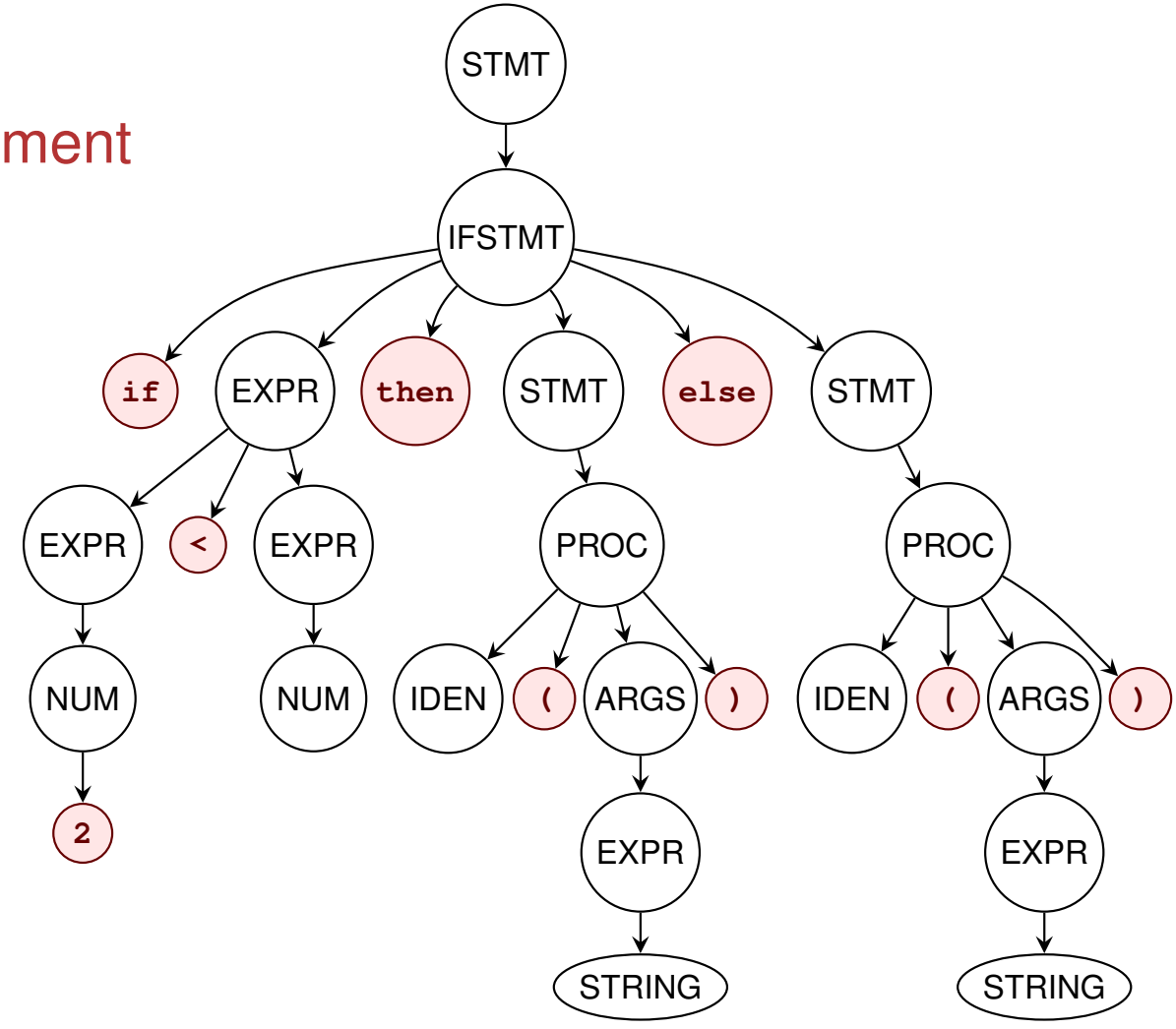


Generated string:

`if` NUM **`<`** NUM **`then`** IDEN **`(`** STRING **`)`** **`else`** IDEN **`(`** STRING **`)`**

Example

A parse tree for an if statement

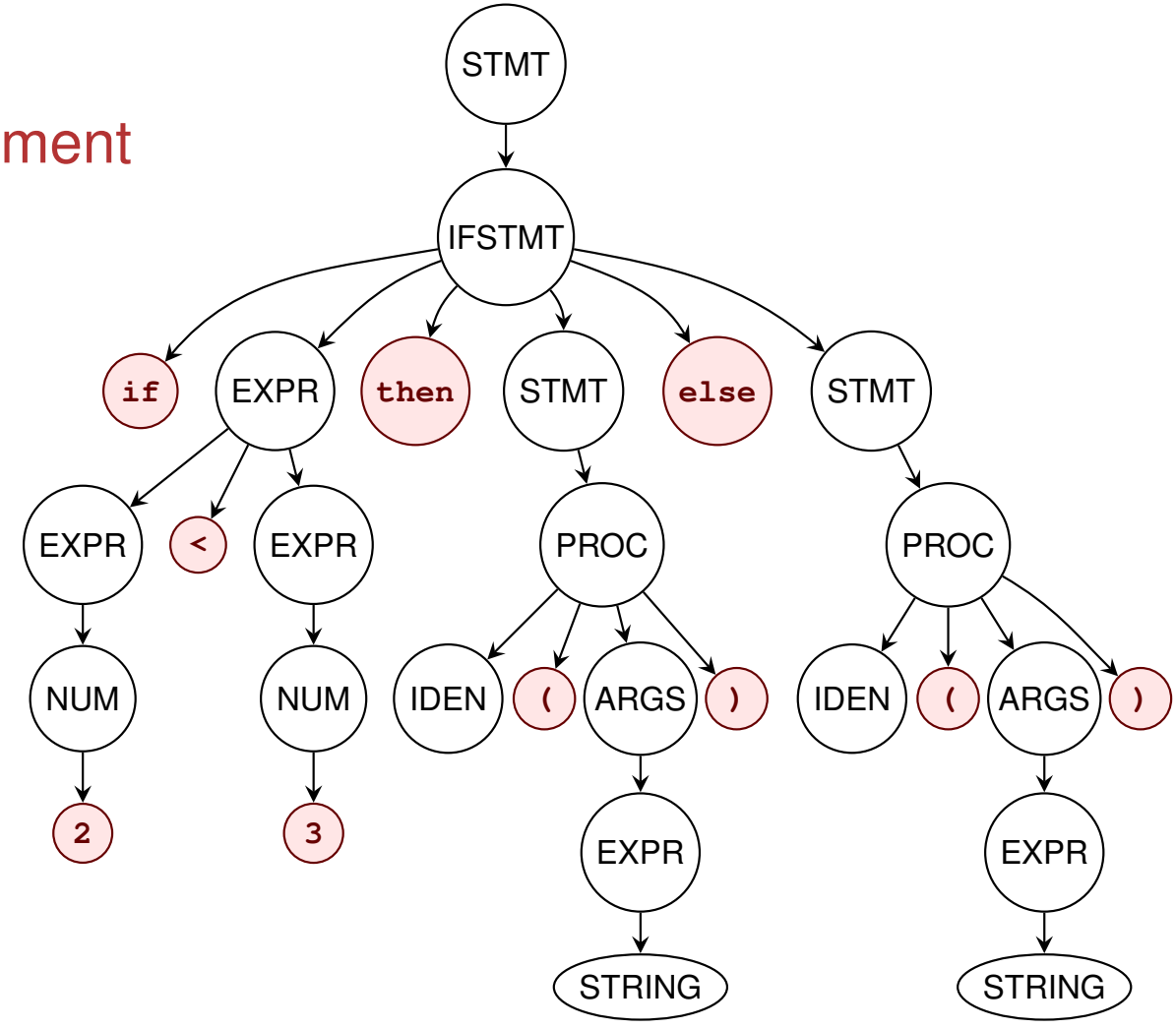


Generated string:

`if 2 <` NUM **`then`** IDEN **`(`** STRING **`)`** **`else`** IDEN **`(`** STRING **`)`**

Example

A parse tree for an if statement

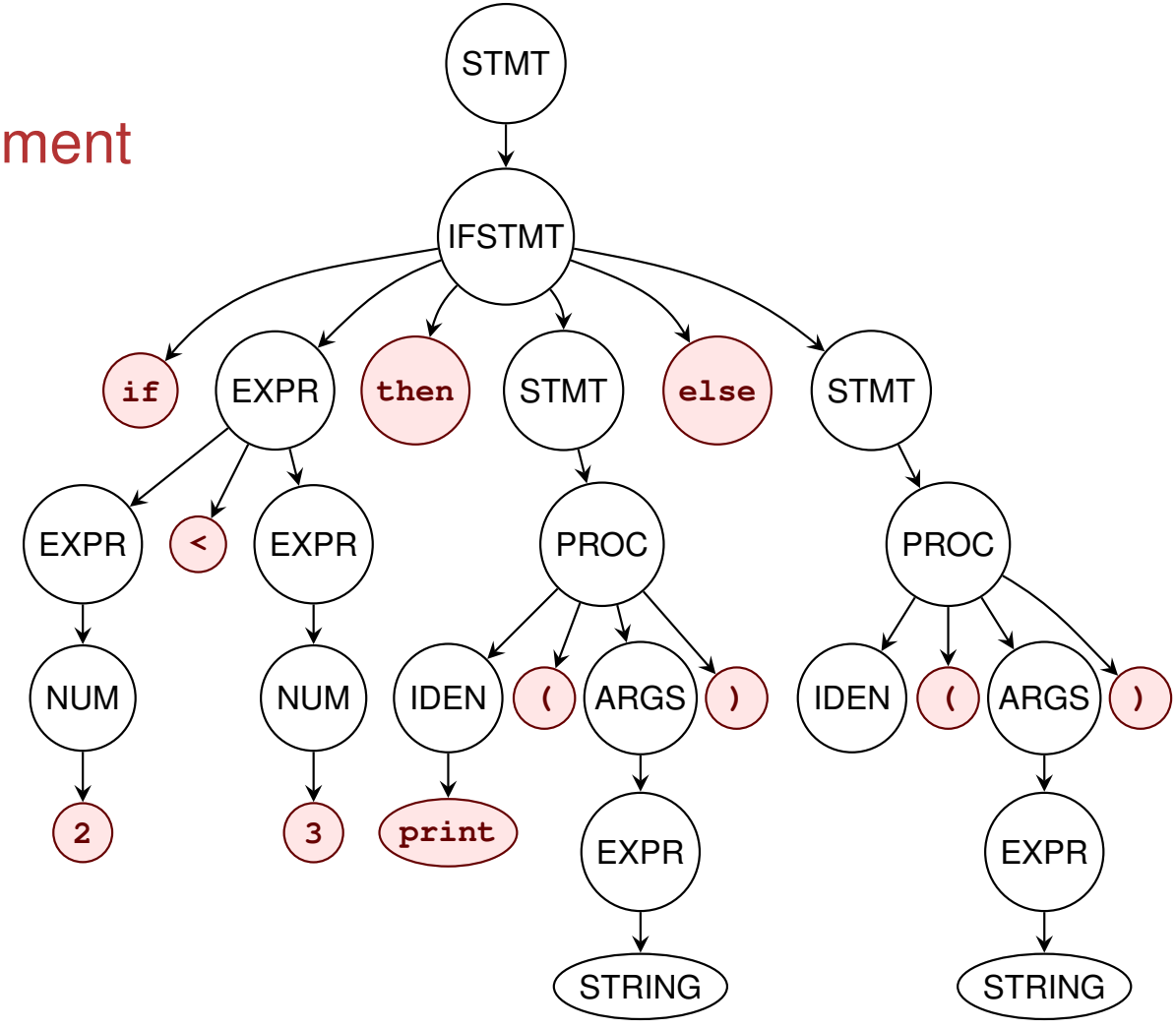


Generated string:

`if 2 < 3 then` IDEN **`(`** STRING **`)`** **`else`** IDEN **`(`** STRING **`)`**

Example

A parse tree for an if statement

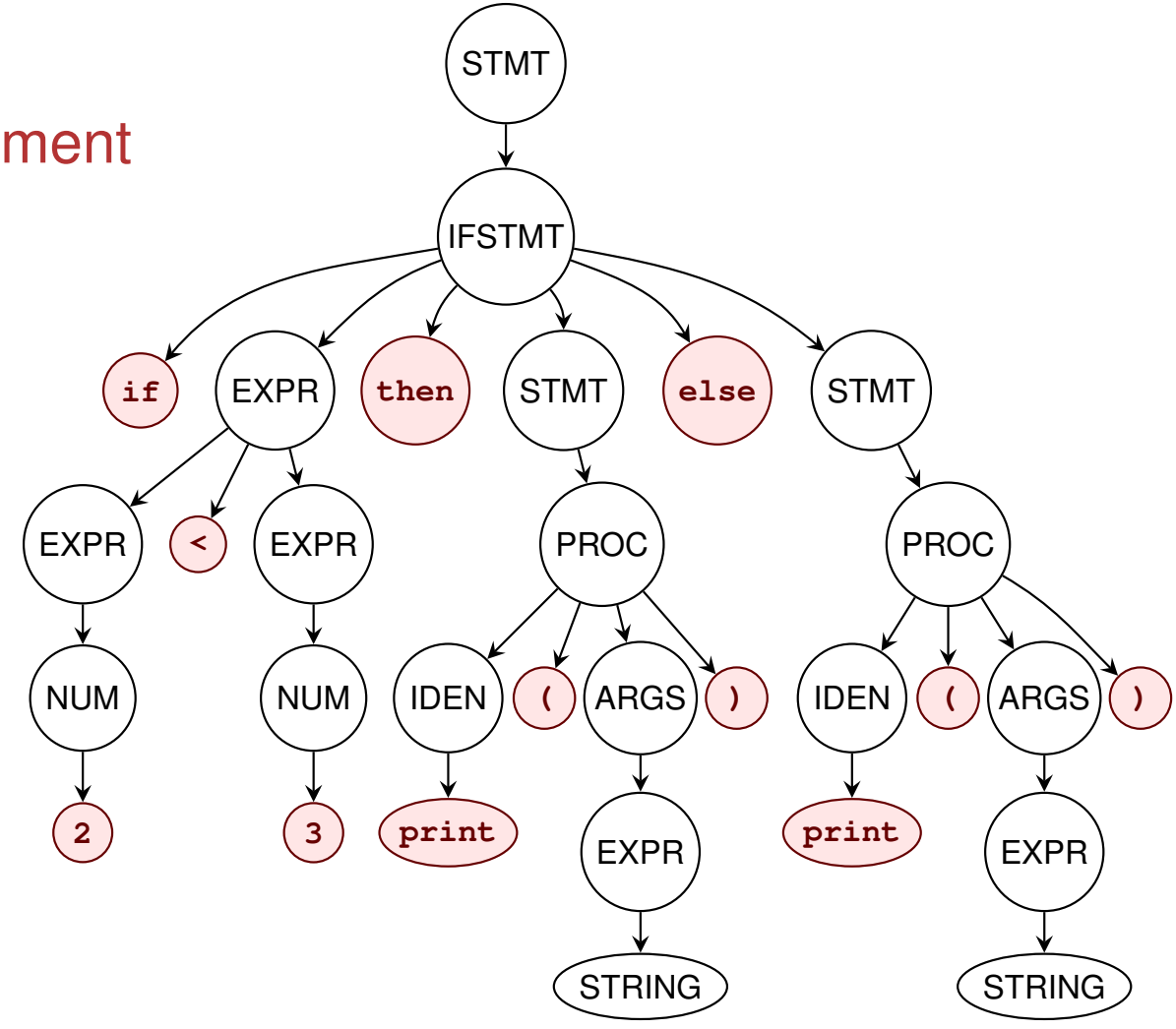


Generated string:

`if 2 < 3 then print (` STRING **`)`** **`else`** IDEN **`(`** STRING **`)`**

Example

A parse tree for an if statement

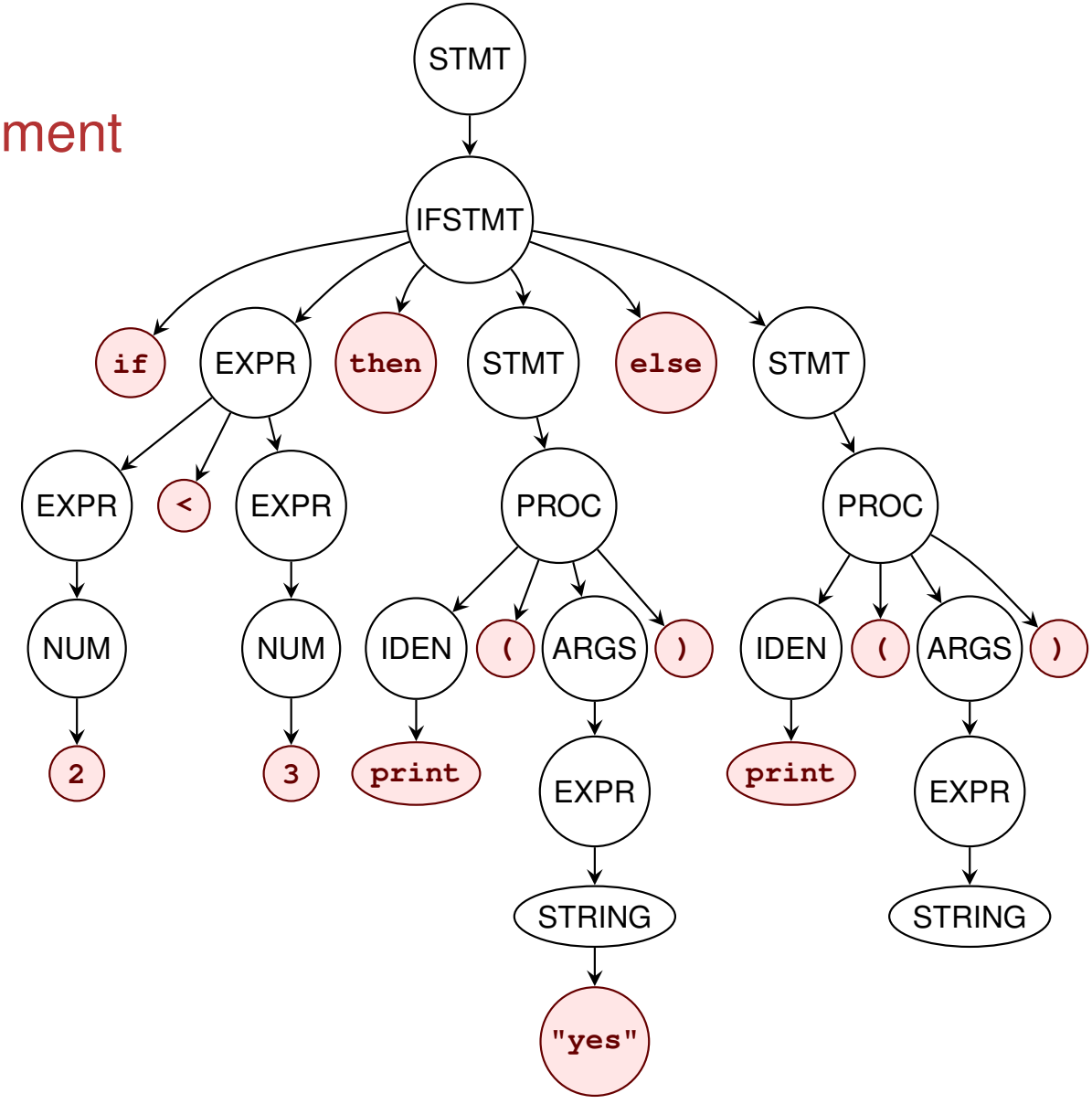


Generated string:

`if 2 < 3 then print (` STRING **`) else print (`** STRING **`)`**

Example

A parse tree for an if statement

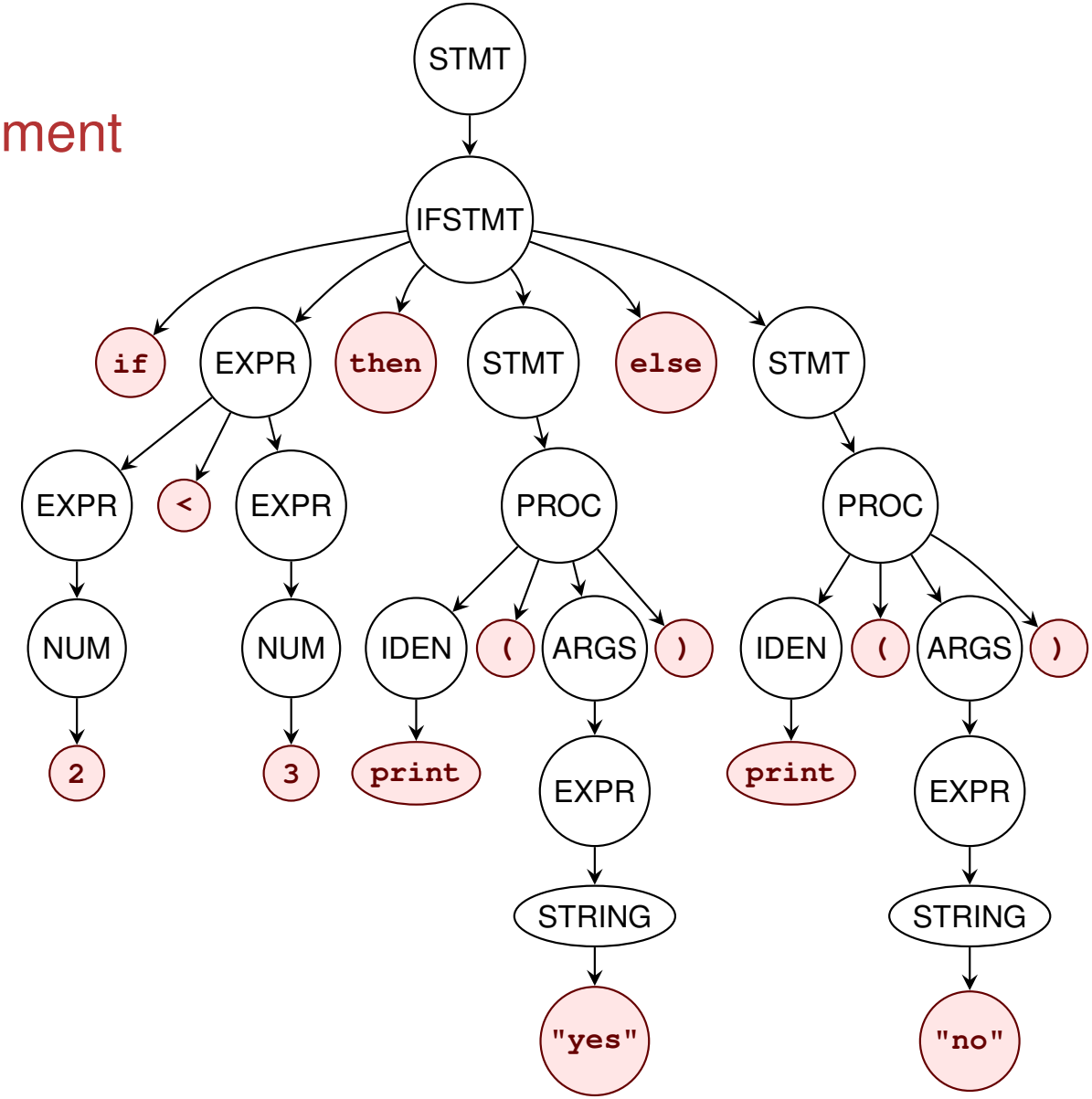


Generated string:

```
if 2 < 3 then print ("yes") else print ( STRING )
```

Example

A parse tree for an if statement



Generated string:

```
if 2 < 3 then print ("yes") else print ("no")
```

Derivations and parse trees

The following is important about parse trees.

- Variable symbols only occur at non-leaf nodes. **(Why?)**

Derivations and parse trees

The following is important about parse trees.

- Variable symbols only occur at non-leaf nodes. **(Why?)**
- Terminal symbols and ε symbols occur at the tree leaves. (When would an ε occur at a leaf?)

Derivations and parse trees

The following is important about parse trees.

- Variable symbols only occur at non-leaf nodes. **(Why?)**
- Terminal symbols and ε symbols occur at the tree leaves. (When would an ε occur at a leaf?)
- An **in-order** traversal of the tree returns the input string (if you remove the ε symbols — **why?**).

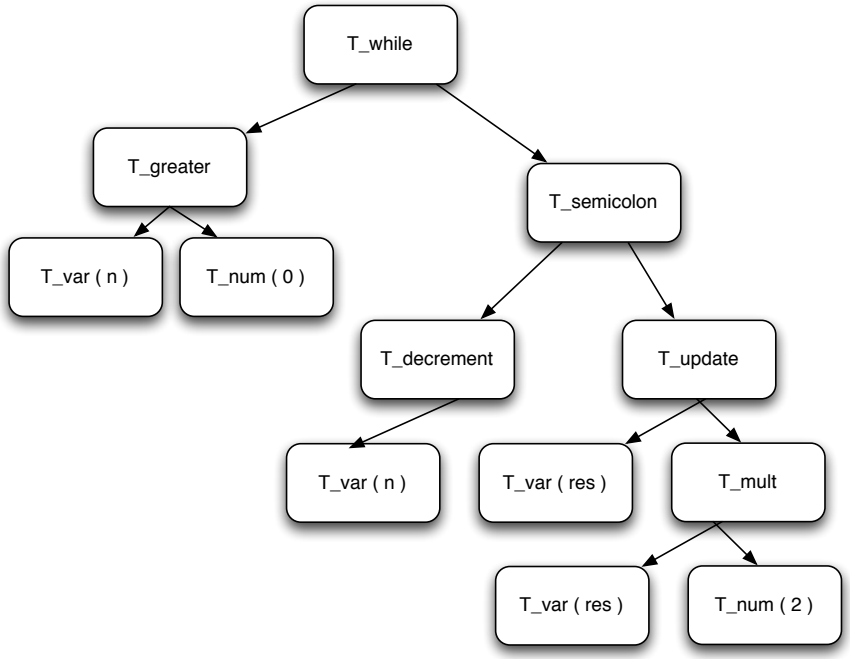
Derivations and parse trees

The following is important about parse trees.

- Variable symbols only occur at non-leaf nodes. **(Why?)**
- Terminal symbols and ε symbols occur at the tree leaves. (When would an ε occur at a leaf?)
- An **in-order** traversal of the tree returns the input string (if you remove the ε symbols — **why?**).
- The parse tree explicitly reveals a particular nesting of structure (*e.g.*, order of evaluation in expressions).

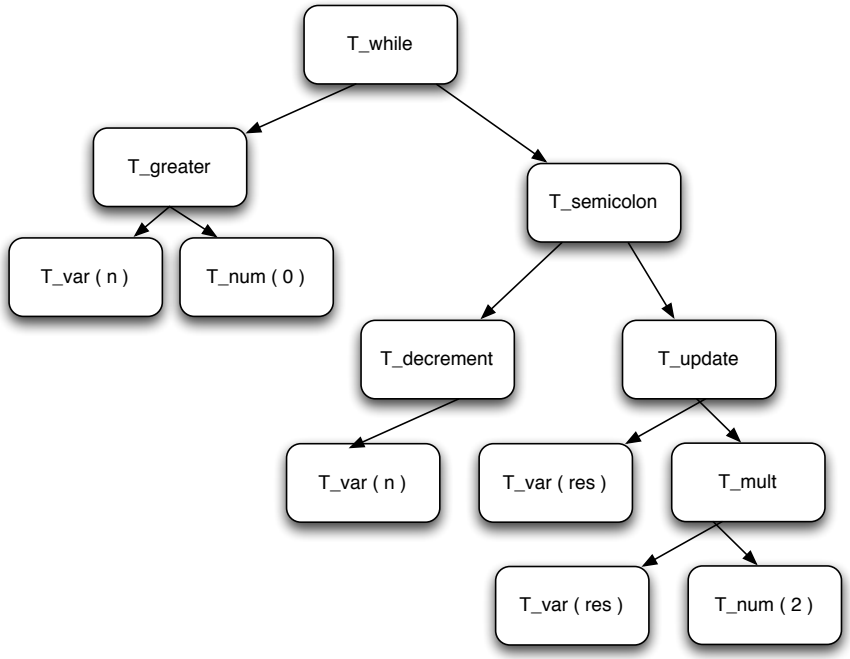
Okay thanks cool story

but *what is an AST??*



Okay thanks cool story

but *what is an AST??*

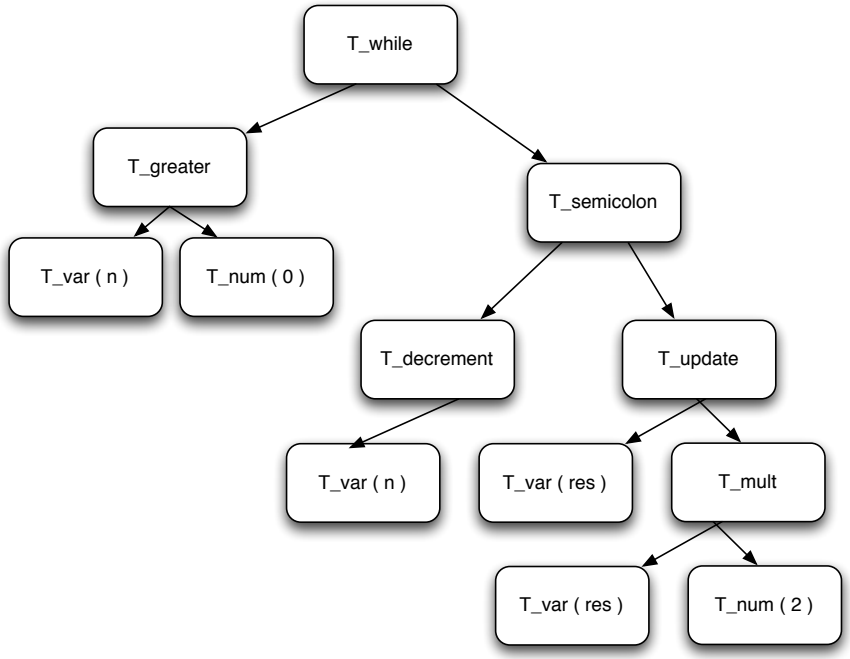


Question. Can we finally talk about what these ASTs are?

What are they??

Okay thanks cool story

but *what is an AST??*



Question. Can we finally talk about what these ASTs are?
What are they??

Answer. A data structure which represents a parse tree.

A simple CFG
— with an AST datatype!

The simple grammar for expressions over integers:

EXPR → NUM | EXPR **+** EXPR | EXPR **−** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

A simple CFG
— with an AST datatype!

The simple grammar for expressions over integers:

EXPR → NUM | EXPR **+** EXPR | EXPR **–** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Java code for an AST for this grammar:

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A simple CFG
— with an AST datatype!

Grammar: EXPR → NUM | EXPR **+** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Java:

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

- The CFG is in effect a specification of the AST datatype.

A simple CFG
— with an AST datatype!

Grammar: EXPR → NUM | EXPR **+** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Java:

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

- The CFG is in effect a specification of the AST datatype.
  - A **`class`** to implement each non-terminal, and each production for that non-terminal

A simple CFG
— with an AST datatype!

Grammar: EXPR → NUM | EXPR **+** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Java:

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

- The CFG is in effect a specification of the AST datatype.
  - A **`class`** to implement each non-terminal, and each production for that non-terminal (except for parentheses, which do not add any meaning to the tree structure)

A simple CFG
— with an AST datatype!

Grammar: EXPR → NUM | EXPR **+** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Java:

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

- The CFG is in effect a specification of the AST datatype.
  - A **`class`** to implement each non-terminal, and each production for that non-terminal (except for parentheses, which do not add any meaning to the tree structure)
  - Relies on **polymorphism** to describe different productions of non-terminals in the AST

A simple CFG

okay very nice thanks — but **where is the AST?!**

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

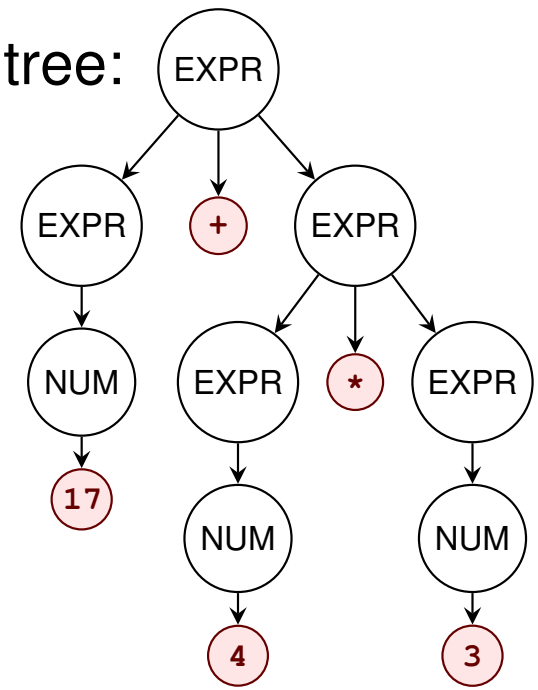
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

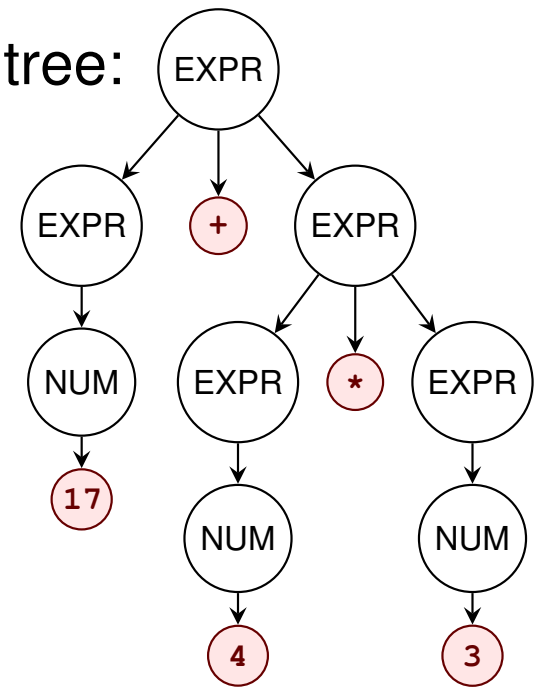
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

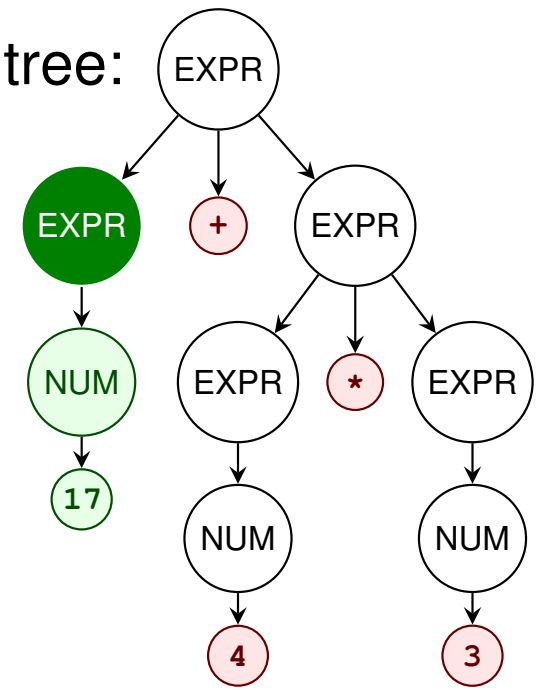
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
   NUM n1 (17);
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

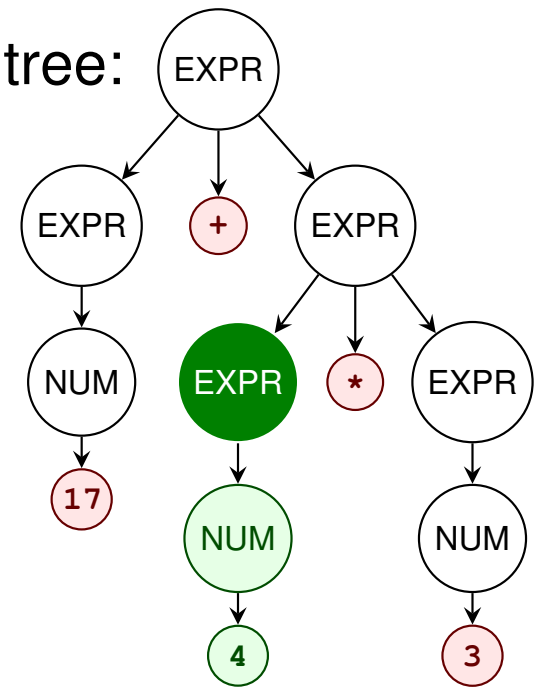
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
    NUM n1 (17);
    NUM n2 (4);
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

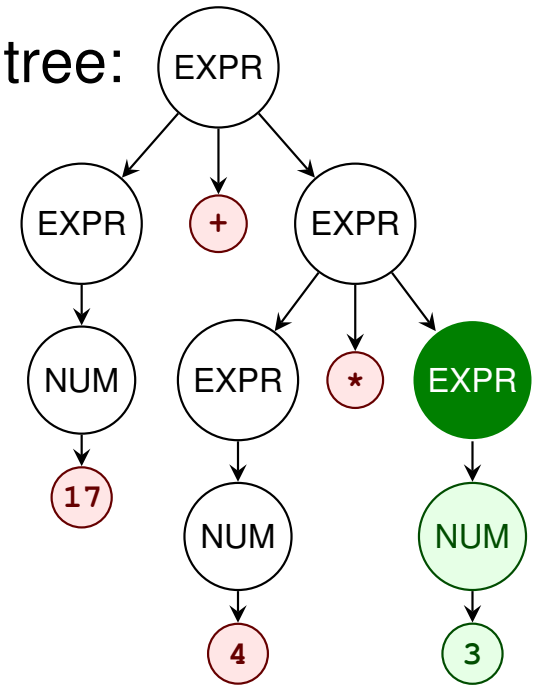
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
   NUM n1 (17);
   NUM n2 (4);
   NUM n3 (3);
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

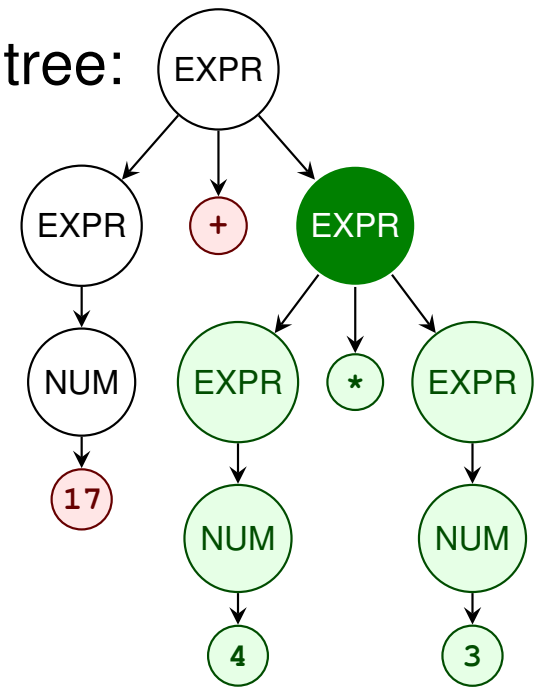
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
   NUM n1 (17);
   NUM n2 (4);
   NUM n3 (3);

   TIMES x (n2, n3);
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

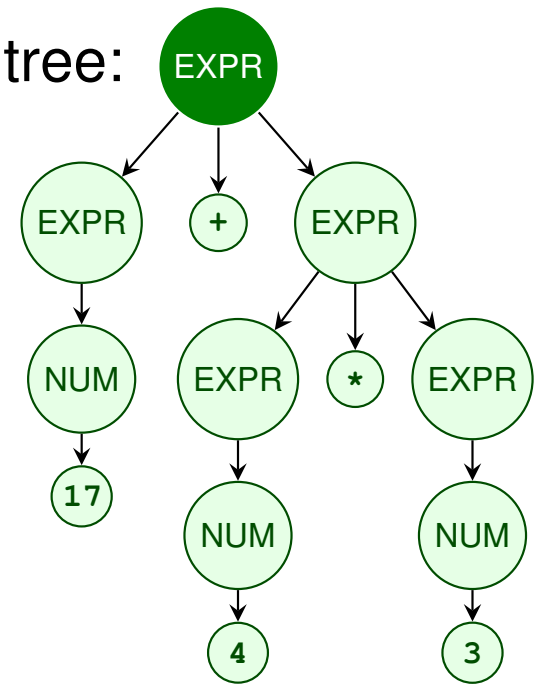
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
   NUM n1 (17);
   NUM n2 (4);
   NUM n3 (3);

   TIMES x (n2, n3);
   ADD p (n1,x);
```

A simple CFG

okay very nice thanks — but **where is the AST?!**

An AST represents a parse tree. You do still have to **build** it.

```
class EXPR {}

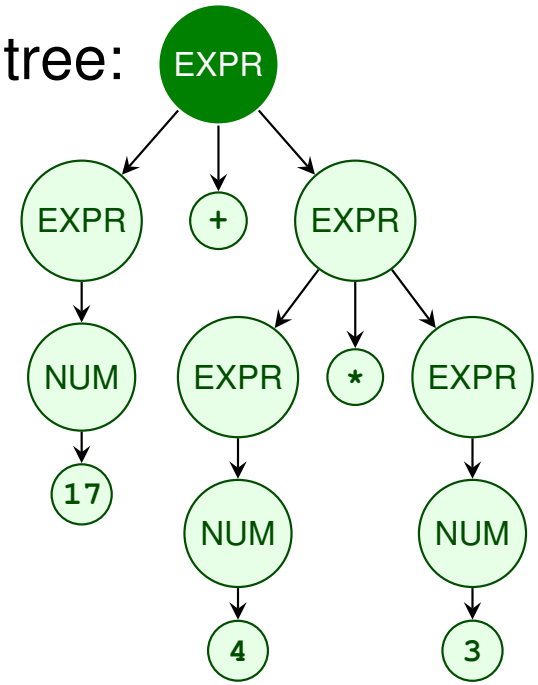
class NUM extends EXPR {
  public int expr;
  NUM (int n) { expr=n; }}

class ADD extends EXPR {
  public EXPR left, right;
  ADD (EXPR a, EXPR b) {
    left=a; right=b; }}
```

```
class DIFF extends EXPR {
  public EXPR left, right;
  DIFF (EXPR a, EXPR b) {
    left=a; right=b; }}

class TIMES extends EXPR {
  public EXPR left, right;
  TIMES (EXPR a, EXPR b) {
    left=a; right=b; }}
```

A parse tree:



Constructing the corresponding AST:

```
{
   NUM n1 (17);
   NUM n2 (4);
   NUM n3 (3);

   TIMES x (n2, n3);
   ADD p (n1,x);
```

— a tree of object/member relationships!

Constructing ASTs

We have finally described what sort of thing an AST is, and how you can construct one:

Constructing ASTs

We have finally described what sort of thing an AST is,
and how you can construct one:

- An AST can be constructed as a recursive tree structure,
  of objects of various classes

Constructing ASTs

We have finally described what sort of thing an AST is,
and how you can construct one:

- An AST can be constructed as a recursive tree structure,
  of objects of various classes
- The classes involved in the AST type
  can be automatically derived from the CFG

Constructing ASTs

We have finally described what sort of thing an AST is,
and how you can construct one:

- An AST can be constructed as a recursive tree structure,
  of objects of various classes
- The classes involved in the AST type
  can be automatically derived from the CFG
- The construction of the AST follows immediately
  from a parse tree for the input program

Constructing ASTs

We have finally described what sort of thing an AST is,
and how you can construct one:

- An AST can be constructed as a recursive tree structure,
  of objects of various classes
- The classes involved in the AST type
  can be automatically derived from the CFG
- The construction of the AST follows immediately
  from a parse tree for the input program

So, it all comes down to constructing a parse tree (finding a derivation) for the program, for the CFG.

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

1. The program has **no** parse trees.

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

1. The program has **no** parse trees.
    - But then the program is not a valid input!

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

1. The program has **no** parse trees.
   - But then the program is not a valid input!
   - We should find a way of giving helpful error messages...
     but for now let's focus on valid programs.

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

1. The program has **no** parse trees.
   - But then the program is not a valid input!
   - We should find a way of giving helpful error messages...
     but for now let's focus on valid programs.

2. The program has **more than one** parse tree.

Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

1. The program has **no** parse trees.
   - But then the program is not a valid input!
   - We should find a way of giving helpful error messages...
     but for now let's focus on valid programs.

2. The program has **more than one** parse tree.
   - — ?! What does that mean?

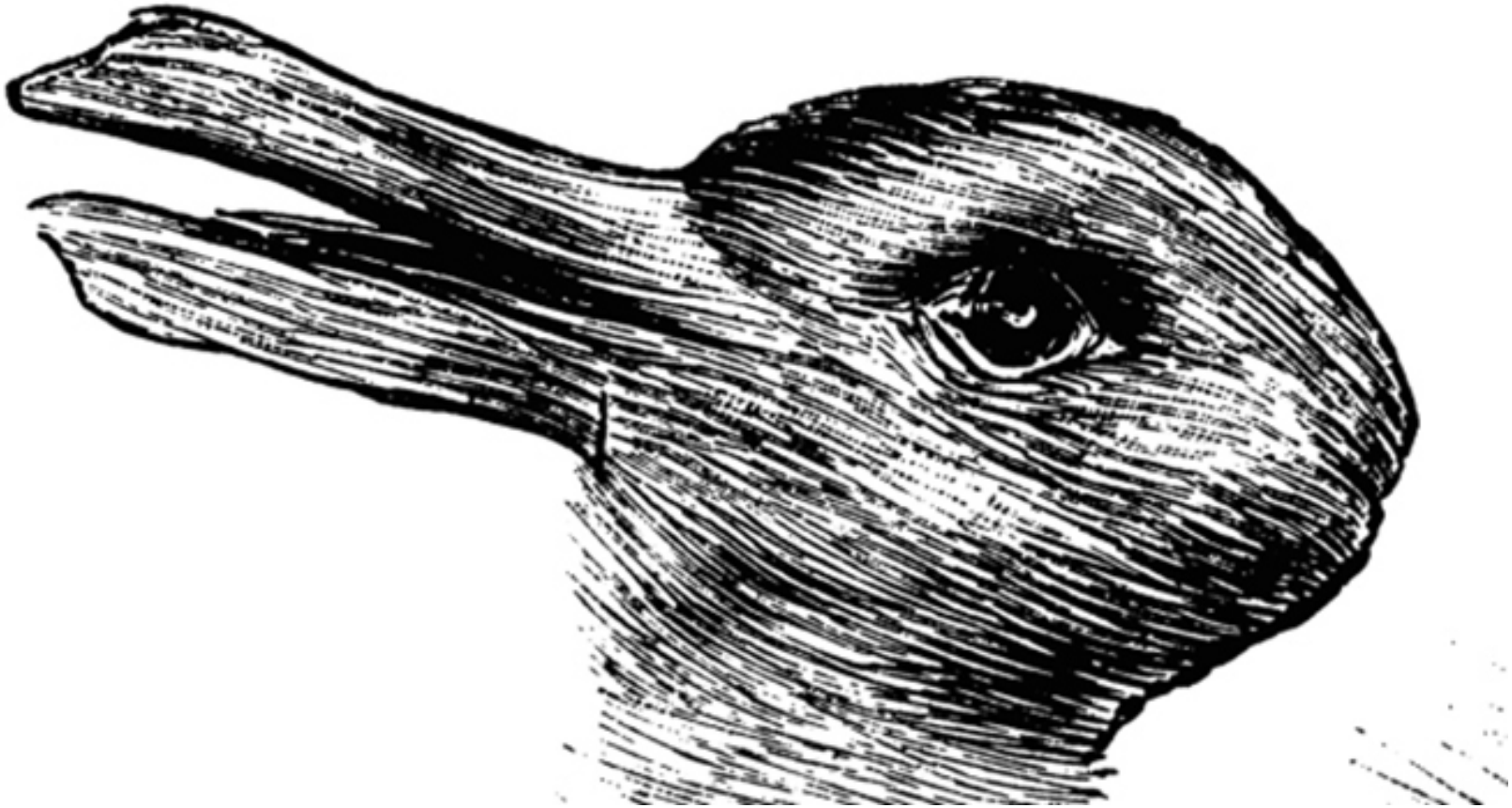
Problems constructing parse trees?

There are two major problems that we can have,
in trying to build a parse tree for an input program.

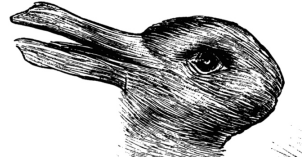
1. The program has **no** parse trees.
   - But then the program is not a valid input!
   - We should find a way of giving helpful error messages...
     but for now let's focus on valid programs.

2. The program has **more than one** parse tree.
   - — ?! What does that mean?
   - That the CFG on its own is **not enough** to determine
     the structure of the input program: it may be interpreted
     in **more than one way**.

Ambiguity

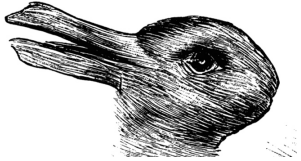


Ambiguity



We have ambiguity in natural language:

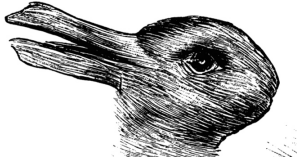
Ambiguity



We have ambiguity in natural language:



Ambiguity

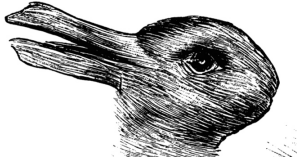


We have ambiguity in natural language:



But surely anything that has to do with **computers**,
is unambiguous — right?

Ambiguity



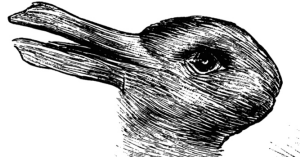
We have ambiguity in natural language:



But surely anything that has to do with **computers**, is unambiguous — right?

- Only if we do **our** jobs well!

Ambiguity



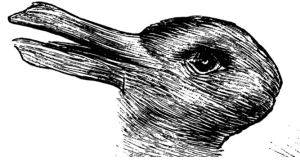
In the grammar

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

the string `17 + 4 * 3` has **two** distinct parse trees!

Ambiguity

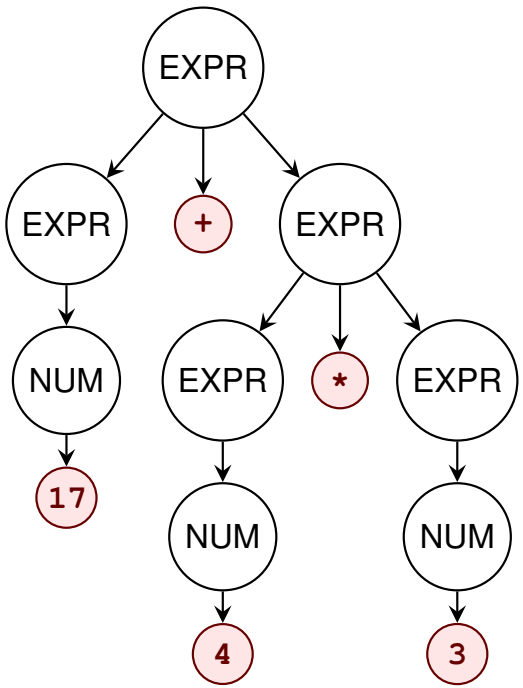


In the grammar

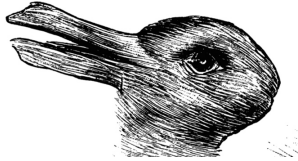
EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

the string `17 + 4 * 3` has **two** distinct parse trees!



Ambiguity

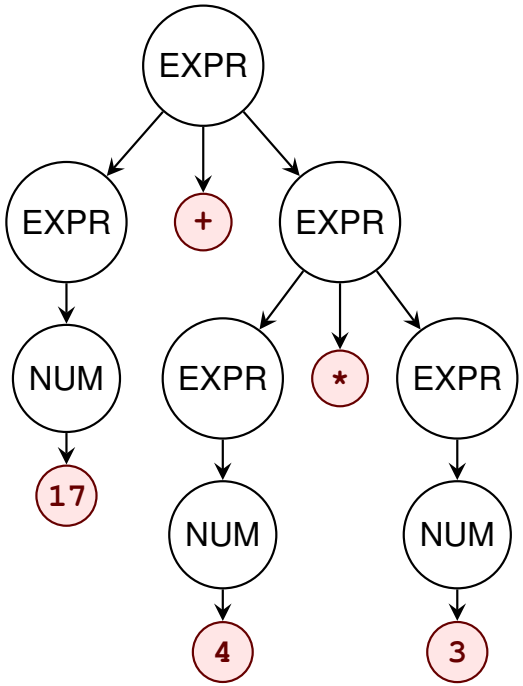


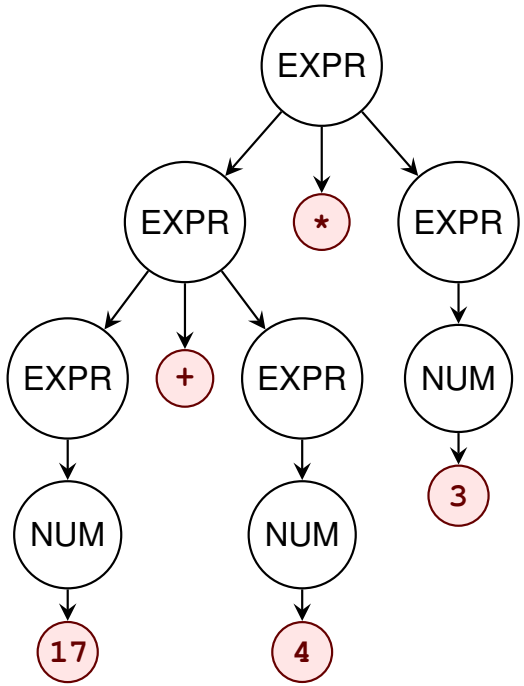
In the grammar

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

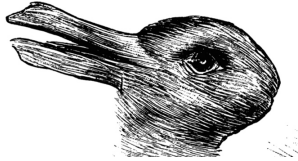
NUM → *(a number)*

the string `17 + 4 * 3` has **two** distinct parse trees!





Ambiguity

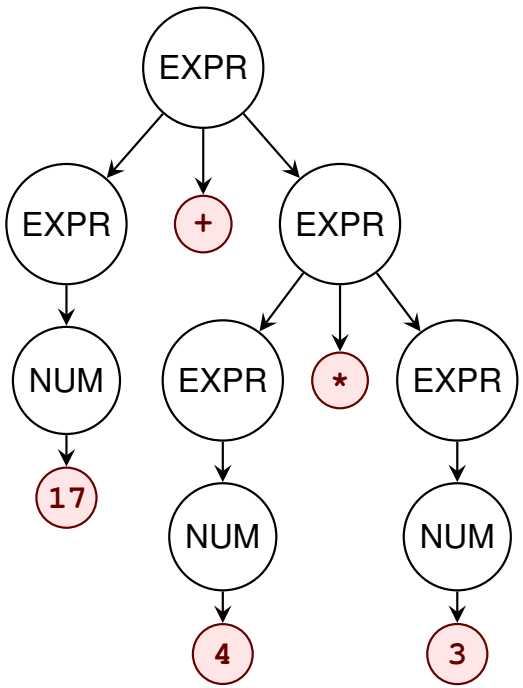


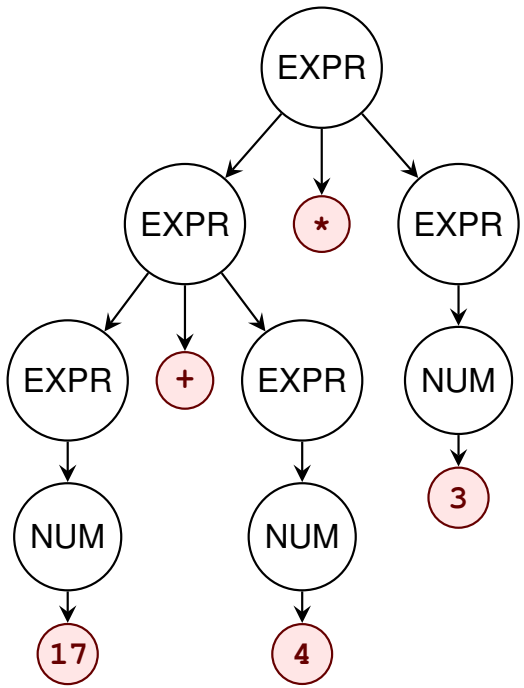
In the grammar

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

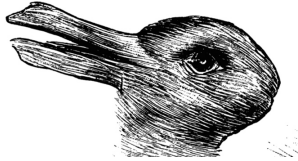
the string `17 + 4 * 3` has **two** distinct parse trees!





The usual interpretations of these parse trees as expressions:

Ambiguity

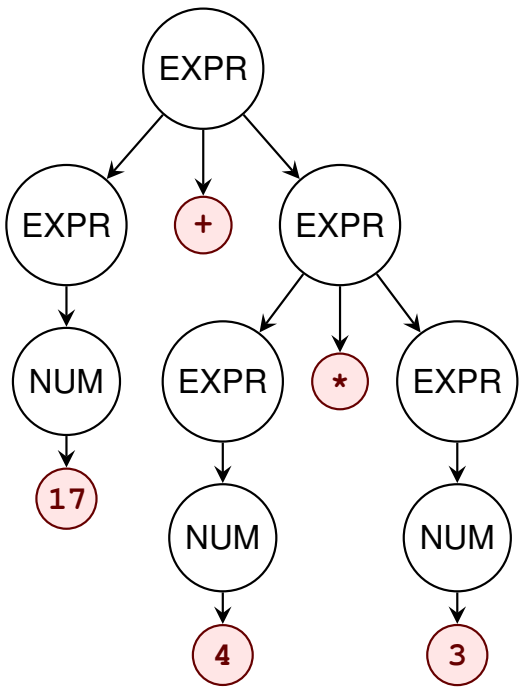


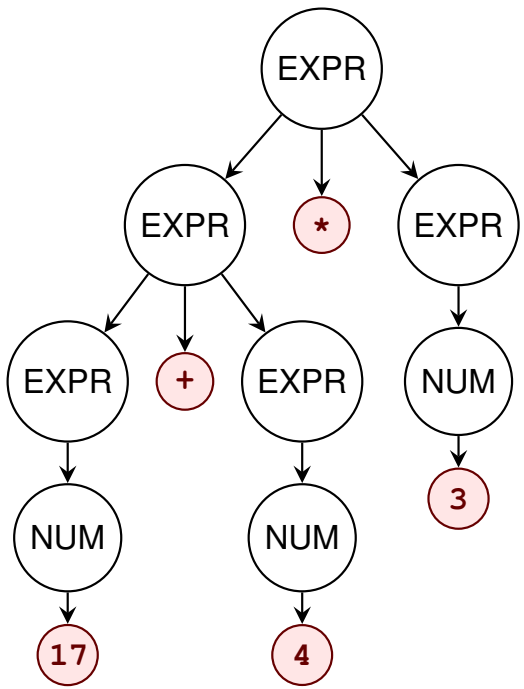
In the grammar

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

the string `17 + 4 * 3` has **two** distinct parse trees!

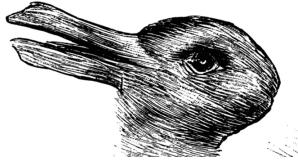




The usual interpretations of these parse trees as expressions:

$$17 + (4 \cdot 3) = 29$$

Ambiguity

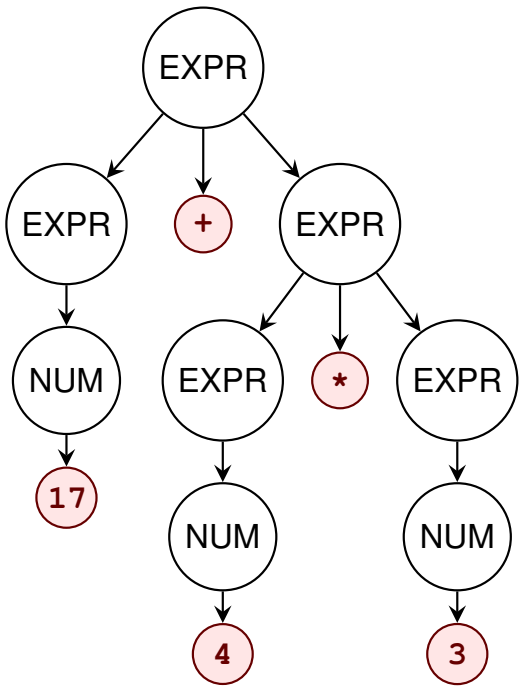


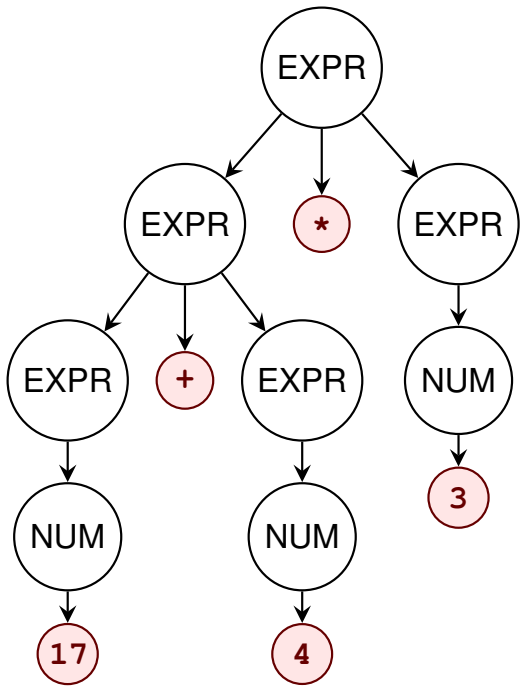
In the grammar

EXPR → NUM | EXPR `+` EXPR | EXPR `-` EXPR | EXPR `*` EXPR | `(` EXPR `)`

NUM → *(a number)*

the string `17 + 4 * 3` has **two** distinct parse trees!

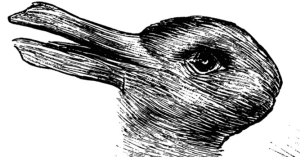




The usual interpretations of these parse trees as expressions:

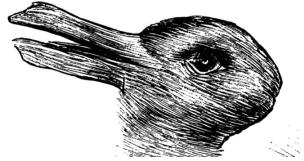
$17 + (4 \cdot 3) = 29$ $\qquad$ $(17 + 4) \cdot 3 = 63$

Ambiguity



We say that a context-free grammar is **ambiguous**,
if there is some string in the language of the grammar,
that has more than one parse tree.

Ambiguity

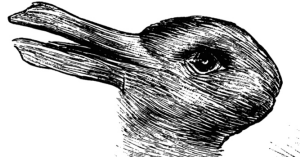


We say that a context-free grammar is **ambiguous**,
if there is some string in the language of the grammar,
that has more than one parse tree.

- This is not the same as having more than one derivation —
  two different derivations can yield the same parse tree

  (if they do “the same” variable substitutions, in different orders)

Ambiguity



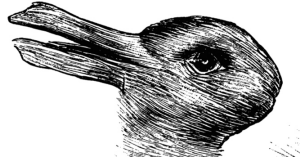
We say that a context-free grammar is **ambiguous**,
if there is some string in the language of the grammar,
that has more than one parse tree.

- This is not the same as having more than one derivation —
  two different derivations can yield the same parse tree

  (if they do “the same” variable substitutions, in different orders)

We say that a language is **inherently ambiguous**,
if there is no “unambiguous” CFG which accepts it.
(Some real programming languages are inherently ambiguous!)

Ambiguity



We say that a context-free grammar is **ambiguous**,
if there is some string in the language of the grammar,
that has more than one parse tree.

- This is not the same as having more than one derivation — two different derivations can yield the same parse tree

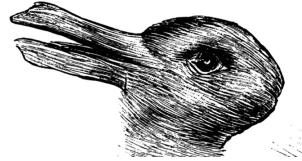
  (if they do “the same” variable substitutions, in different orders)

We say that a language is **inherently ambiguous**,
if there is no “unambiguous” CFG which accepts it.
(Some real programming languages are inherently ambiguous!)

Ambiguity in programming languages is **dangerous**,
and must be addressed in some way

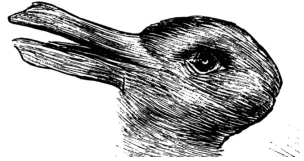
- it leaves the meaning of a program unclear — to the programmer, but also **to the compiler writer**

Ambiguity



What can we do about ambiguity
— if resolving the ambiguity is complicated?

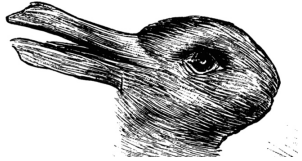
Ambiguity



What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?

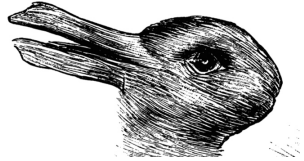
Ambiguity



What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about

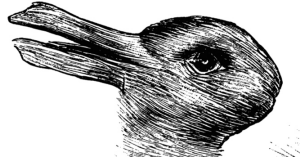
Ambiguity



What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about
  - And what would you do if you found out your language was ambiguous, anyway?

Ambiguity

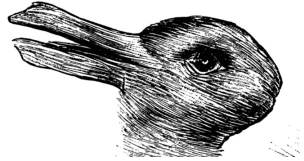


What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about
  - And what would you do if you found out your language was ambiguous, anyway?

- Have the parser find / construct all possible parse trees?

Ambiguity

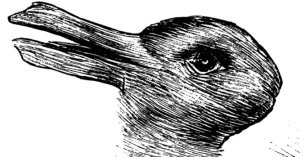


What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about
  - And what would you do if you found out your language was ambiguous, anyway?

- Have the parser find / construct all possible parse trees?
  - The “Earley parser” does this

Ambiguity

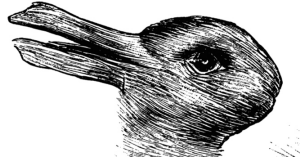


What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about
  - And what would you do if you found out your language was ambiguous, anyway?

- Have the parser find / construct all possible parse trees?
  - The “Earley parser” does this
  - Upside: later compiler stages could determine which to use

Ambiguity

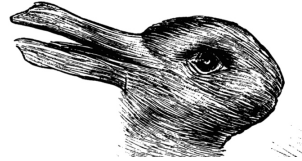


What can we do about ambiguity
— if resolving the ambiguity is complicated?

- Automatically check whether a grammar is ambiguous?
  - Bad news: ambiguity of grammars is **undecidable**
    — it is literally impossible to write an algorithm to do it, with any technique that we know about
  - And what would you do if you found out your language was ambiguous, anyway?

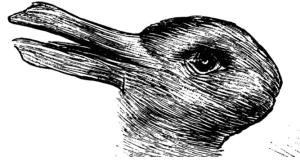
- Have the parser find / construct all possible parse trees?
  - The “Earley parser” does this
  - Upside: later compiler stages could determine which to use
  - Downside: requires a lot of extra work for parse trees which will ultimately be discarded.

Ambiguity



What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

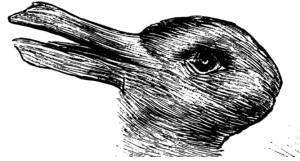
Ambiguity



What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

- Add extra conventions? (*e.g.*, priority of reduction rules: "$*$ binds more tightly than $+$")

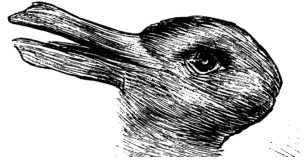
Ambiguity



What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

- Add extra conventions? (*e.g.*, priority of reduction rules: "`*` binds more tightly than `+`")
  - Upside: a simple idea which is not *too* difficult in practise

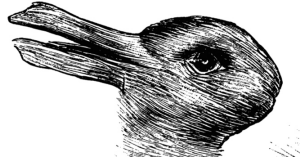
Ambiguity



What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

- Add extra conventions? (*e.g.*, priority of reduction rules:
  "`*` binds more tightly than `+`")
  - Upside: a simple idea which is not *too* difficult in practise
  - Downside: requires care in implementation,
    and added emphasis in the language spec

Ambiguity

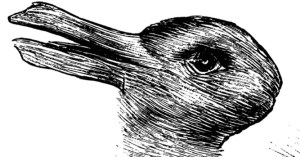


What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

- Add extra conventions? (*e.g.*, priority of reduction rules:
  "`*` binds more tightly than `+`")
  - Upside: a simple idea which is not *too* difficult in practise
  - Downside: requires care in implementation,
    and added emphasis in the language spec

- Rewrite the grammar to remove ambiguity?

Ambiguity

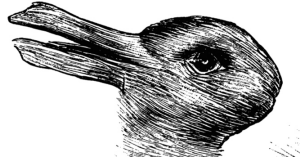


What can we do about ambiguity
— if we have a good idea of how to resolve ambiguous cases?

- Add extra conventions? (*e.g.*, priority of reduction rules: "`*` binds more tightly than `+`")
  - Upside: a simple idea which is not *too* difficult in practise
  - Downside: requires care in implementation, and added emphasis in the language spec

- Rewrite the grammar to remove ambiguity?
  - Often possible, and practical to do
  - Explicitly solves the problem

Ambiguity: grammar rewriting



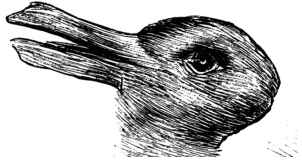
The problem with

EXPR → NUM | EXPR + EXPR | EXPR − EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

is that addition and multiplication have the same status.

Ambiguity: grammar rewriting



The problem with

EXPR → NUM | EXPR **+** EXPR | EXPR **–** EXPR | EXPR ***** EXPR | **(** EXPR **)**

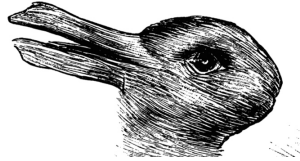
NUM → *(a number)*

is that addition and multiplication have the same status.
But the convention that **we** use ourselves, is to evaluate

$$a + b \cdot c = a + (b \cdot c)$$ "times before plus"

$$a - b - c = (a - b) - c$$ evaluate left-to-right
. . . more important for subtraction, division

Ambiguity: grammar rewriting



The problem with

EXPR → NUM | EXPR + EXPR | EXPR − EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

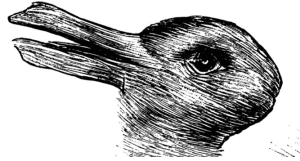
is that addition and multiplication have the same status.
But the convention that **we** use ourselves, is to evaluate

$$a + b \cdot c = a + (b \cdot c)$$ "times before plus"

$$a - b - c = (a - b) - c$$ evaluate left-to-right
. . . more important for subtraction, division

On it's own, this grammar doesn't provide any such distinctions.

Ambiguity: grammar rewriting



The problem with

EXPR → NUM | EXPR + EXPR | EXPR − EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

is that addition and multiplication have the same status.
But the convention that **we** use ourselves, is to evaluate

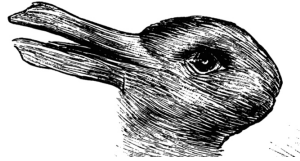
$$a + b \cdot c = a + (b \cdot c)$$ "times before plus"

$$a - b - c = (a - b) - c$$ evaluate left-to-right
. . . more important for subtraction, division

On it's own, this grammar doesn't provide any such distinctions.

- Prioritising reductions:
  - Information **beyond** the CFG specification.
  - Necessary for inherently ambiguous languages

Ambiguity: grammar rewriting



The problem with

EXPR → NUM | EXPR + EXPR | EXPR − EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

is that addition and multiplication have the same status.
But the convention that **we** use ourselves, is to evaluate

$$a + b \cdot c = a + (b \cdot c)$$ "times before plus"

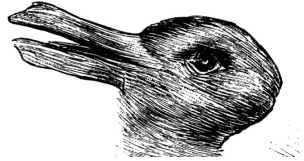
$$a - b - c = (a - b) - c$$ evaluate left-to-right

. . . more important for subtraction, division

On it's own, this grammar doesn't provide any such distinctions.

- Prioritising reductions:
  - Information **beyond** the CFG specification.
  - Necessary for inherently ambiguous languages
- Here, we add the conventions **to the grammar itself**

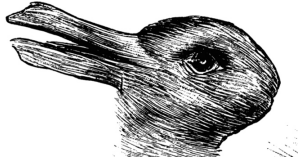
Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

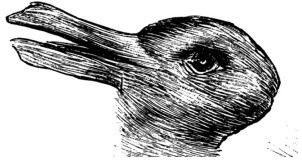
EXPR → TERM | TERM + EXPR | TERM – EXPR

TERM → FACT | FACT * TERM

FACT → NUM | (EXPR)

NUM → *(a number)*

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

EXPR → TERM | TERM + EXPR | TERM – EXPR

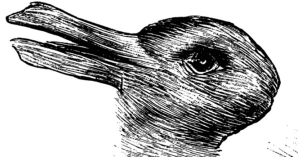
TERM → FACT | FACT * TERM

FACT → NUM | (EXPR)

NUM → *(a number)*

- Eliminates ambiguity
  - Multiplication now does “bind more tightly” than addition

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

EXPR → TERM | TERM + EXPR | TERM – EXPR

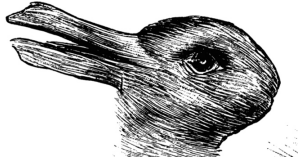
TERM → FACT | FACT * TERM

FACT → NUM | (EXPR)

NUM → *(a number)*

- Eliminates ambiguity
  - Multiplication now does "bind more tightly" than addition
- But it makes addition, multiplication, **and subtraction** group to the right (*"right-associative"*)!

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

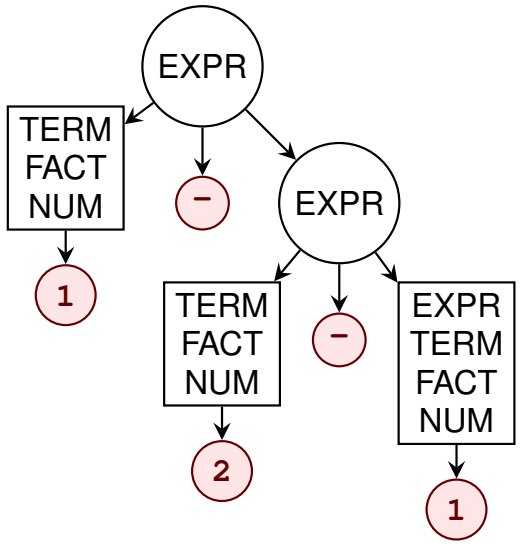
NUM → *(a number)*

EXPR → TERM | TERM + EXPR | TERM – EXPR

TERM → FACT | FACT * TERM

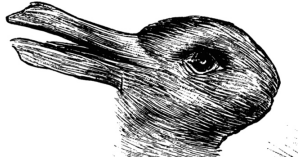
FACT → NUM | (EXPR)

NUM → *(a number)*



- Eliminates ambiguity
  - Multiplication now does “bind more tightly” than addition
- But it makes addition, multiplication, **and subtraction** group to the right (*“right-associative”*)!

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

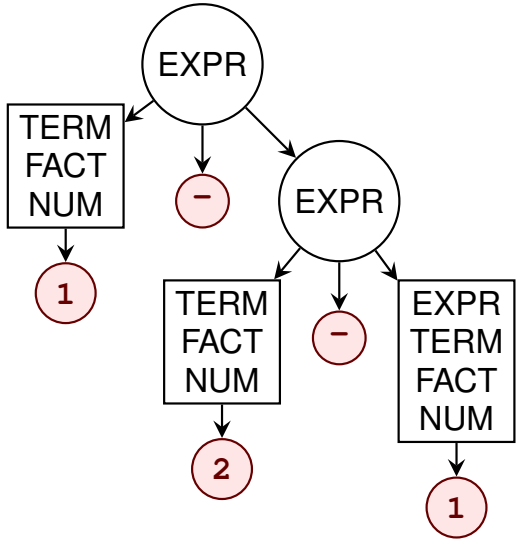
NUM → *(a number)*

EXPR → TERM | TERM + EXPR | TERM – EXPR

TERM → FACT | FACT * TERM

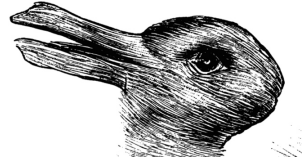
FACT → NUM | (EXPR)

NUM → *(a number)*



- Eliminates ambiguity
  - Multiplication now does “bind more tightly” than addition
- But it makes addition, multiplication, **and subtraction** group to the right (*“right-associative”*)!

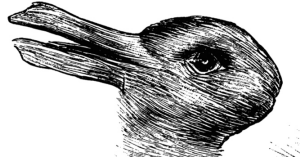
Ambiguity: grammar rewriting



EXPR → NUM | EXPR **+** EXPR | EXPR **–** EXPR | EXPR ***** EXPR | **(** EXPR **)**

NUM → *(a number)*

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

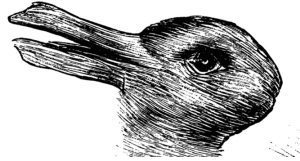
EXPR → TERM | EXPR + TERM | EXPR – TERM

TERM → FACT | TERM * FACT

FACT → NUM | (EXPR)

NUM → *(a number)*

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

NUM → *(a number)*

EXPR → TERM | EXPR + TERM | EXPR – TERM

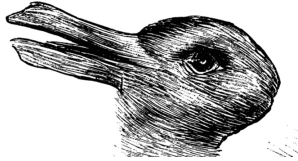
TERM → FACT | TERM * FACT

FACT → NUM | (EXPR)

NUM → *(a number)*

- Eliminates ambiguity
- Makes addition, multiplication, and subtraction group to the left

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

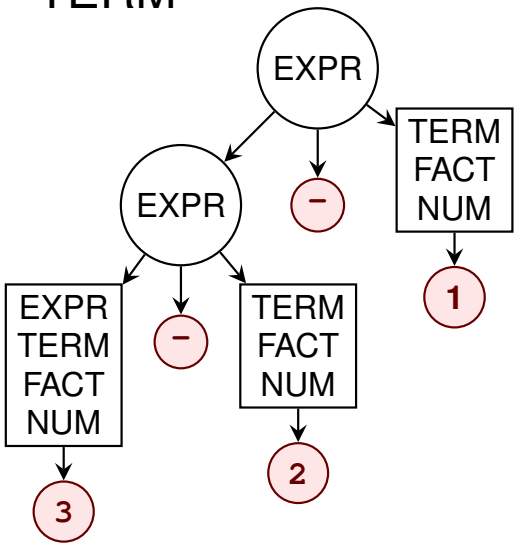
NUM → *(a number)*

EXPR → TERM | EXPR + TERM | EXPR – TERM

TERM → FACT | TERM * FACT

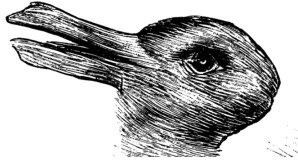
FACT → NUM | (EXPR)

NUM → *(a number)*



- Eliminates ambiguity
- Makes addition, multiplication, and subtraction group to the left

Ambiguity: grammar rewriting



EXPR → NUM | EXPR + EXPR | EXPR – EXPR | EXPR * EXPR | (EXPR)

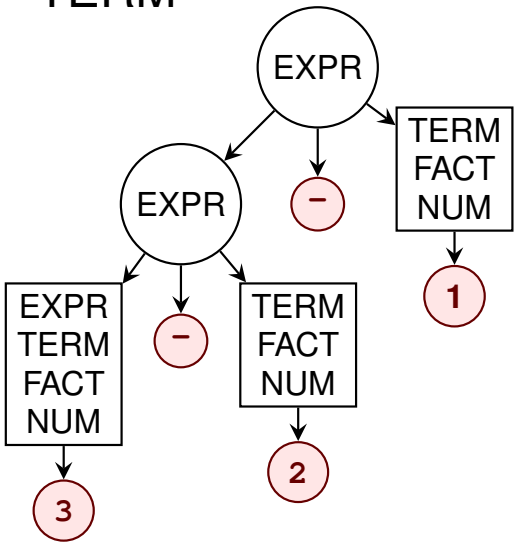
NUM → *(a number)*

EXPR → TERM | EXPR + TERM | EXPR – TERM

TERM → FACT | TERM * FACT

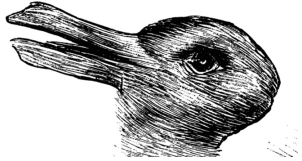
FACT → NUM | (EXPR)

NUM → *(a number)*



- Eliminates ambiguity
- Makes addition, multiplication, and subtraction group to the left
- This grammar is **left-recursive** — there are multi-step reductions such that $\mathtt{x} \Rightarrow^* \mathtt{xy}$
  - A technical problem for some approaches to parsing
  - We'll see how to fix this in future lectures

If-Then ambiguity

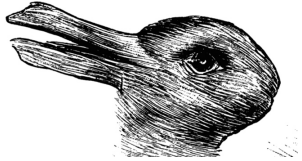


Here is a problem that often arises when specifying programming languages.

Suppose that we have a grammar including:

$$M \to \texttt{if}\ E\ \texttt{then}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \cdots$$

If-Then ambiguity



Here is a problem that often arises when specifying programming languages.

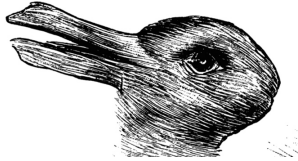
Suppose that we have a grammar including:

$$M \to \texttt{if}\ E\ \texttt{then}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \cdots$$

Now we can find two distinct parse trees for

```
if B then if B' then P else Q
```

If-Then ambiguity



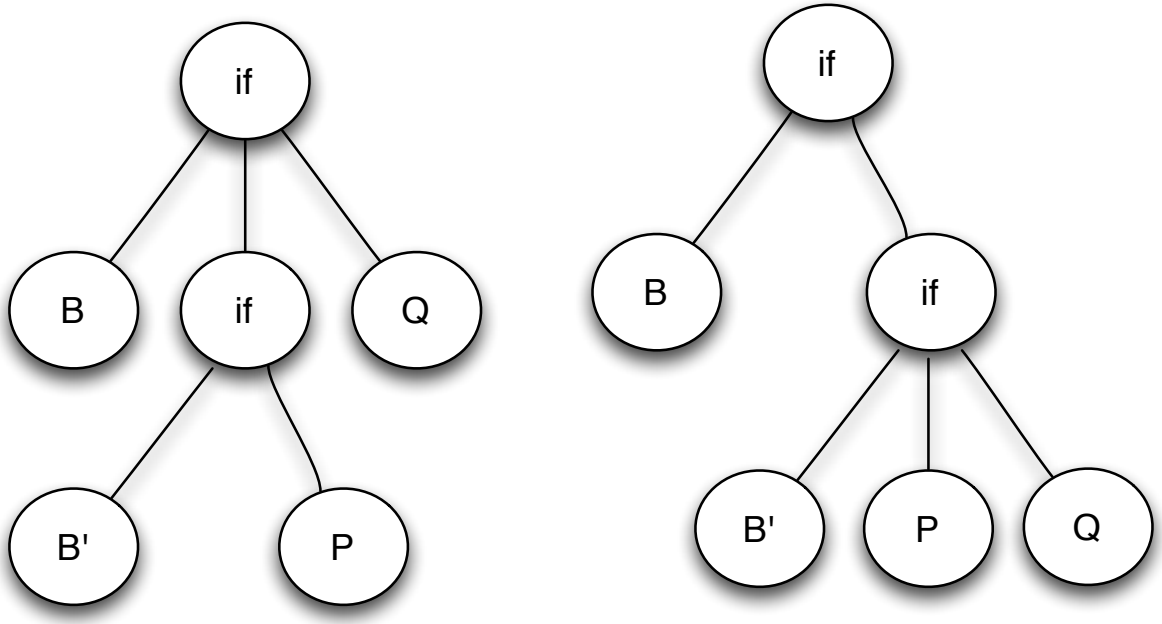
Here is a problem that often arises when specifying programming languages.

Suppose that we have a grammar including:

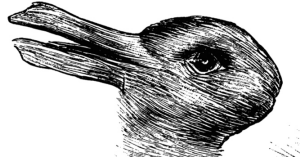
$$M \to \texttt{if}\ E\ \texttt{then}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \cdots$$

Now we can find two distinct parse trees for

```
if B then if B' then P else Q
```

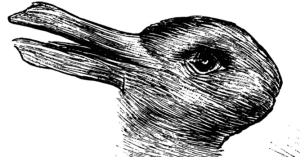


If-Then ambiguity



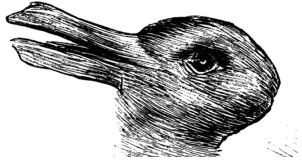
Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses!

If-Then ambiguity



Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses! — prioritising “if-then-else” over “if-then”.

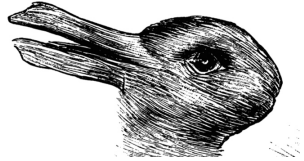
If-Then ambiguity



Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses! — prioritising “if-then-else” over “if-then”.

As with prioritising multiplication over addition, we make “closed” **`if`** statements bind more tightly than “open” ones:

If-Then ambiguity

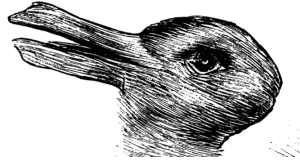


Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses! — prioritising "if-then-else" over "if-then".

As with prioritising multiplication over addition, we make "closed" **`if`** statements bind more tightly than "open" ones:

$$
\begin{array}{lcll}
M & \rightarrow & \mathit{OPEN_IF} & \textit{// some 'if's have no 'else's} \\
 & | & \mathit{CL_IF} & \textit{// only 'if's with 'else's}
\end{array}
$$

If-Then ambiguity

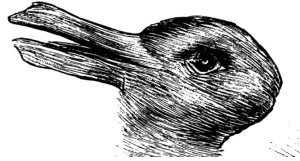


Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses! — prioritising "if-then-else" over "if-then".

As with prioritising multiplication over addition, we make "closed" **`if`** statements bind more tightly than "open" ones:

$$
\begin{array}{rcll}
M & \rightarrow & \mathit{OPEN_IF} & \text{// some 'if's have no 'else's} \\
 & | & \mathit{CL_IF} & \text{// only 'if's with 'else's} \\
 & & & \\
\mathit{OPEN_IF} & \rightarrow & \texttt{if}\ E\ \texttt{then}\ M & \text{// 'if' without matching 'else'} \\
 & | & \texttt{if}\ E\ \texttt{then}\ \mathit{CL_IF}\ \texttt{else}\ \mathit{OPEN_IF} & \text{// 'inner if's are closed}
\end{array}
$$

If-Then ambiguity

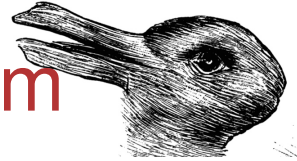


Suppose that we adopt the convention that every **`else`** binds with the most recent unclosed **`if`** — like parentheses! — prioritising "if-then-else" over "if-then".

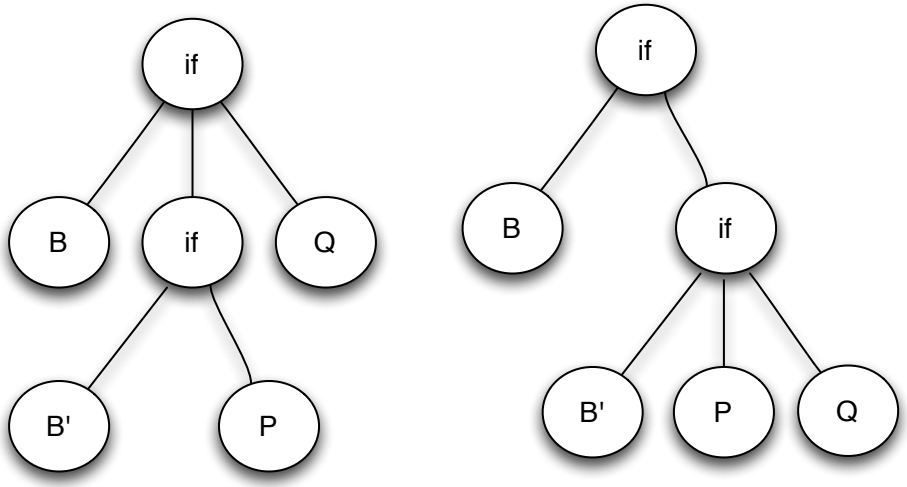
As with prioritising multiplication over addition, we make "closed" **`if`** statements bind more tightly than "open" ones:

M	→	*OPEN_IF*		*// some 'if's have no 'else's*
	|	*CL_IF*		*// only 'if's with 'else's*
OPEN_IF	→	**`if`** *E* **`then`** *M*		*// 'if' without matching 'else'*
	|	**`if`** *E* **`then`** *CL_IF* **`else`**	*OPEN_IF*	*// 'inner if's are closed*
CL_IF	→	**`if`** *E* **`then`** *CL_IF* **`else`**	*CL_IF*	*// nicely closed 'if's*
	|	*NON_IF_STMTS*		*// no 'if's at all*

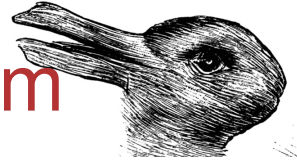
If-Then ambiguity, aka the dangling-else problem



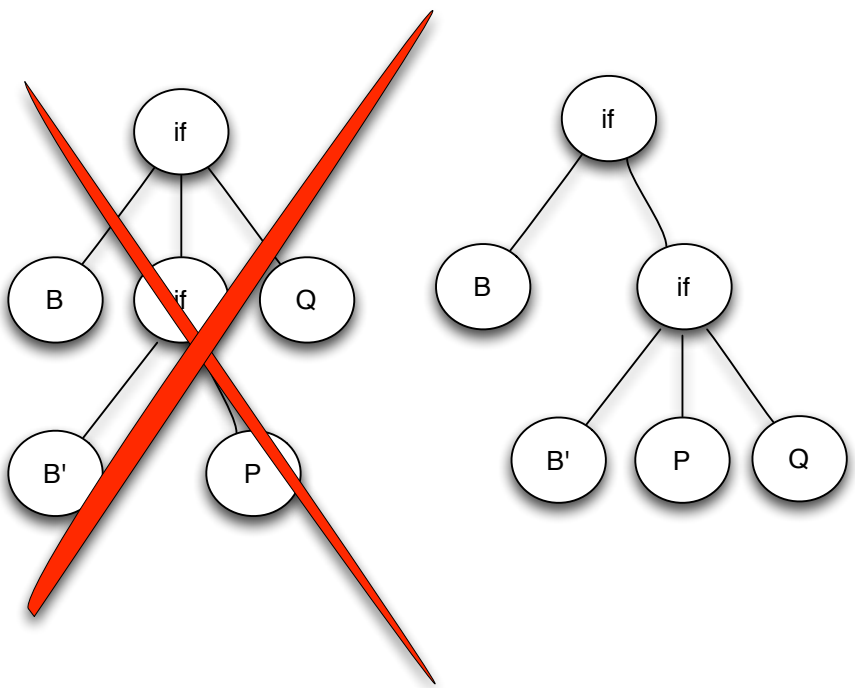
$$
\begin{array}{rcll}
M & \rightarrow & \mathit{OPEN_IF} & \text{// some 'if's have no 'else's} \\
 & | & \mathit{CL_IF} & \text{// only 'if's with 'else's} \\
 & & & \\
\mathit{OPEN_IF} & \rightarrow & \texttt{if}\ E\ \texttt{then}\ M & \text{// 'if' without matching 'else'} \\
 & | & \texttt{if}\ E\ \texttt{then}\ \mathit{CL_IF}\ \texttt{else}\ \mathit{OPEN_IF} & \text{// 'inner if's are closed} \\
 & & & \\
\mathit{CL_IF} & \rightarrow & \texttt{if}\ E\ \texttt{then}\ \mathit{CL_IF}\ \texttt{else}\ \mathit{CL_IF} & \text{// nicely closed 'if's} \\
 & | & \mathit{NON_IF_STMTS} & \text{// no 'if's at all}
\end{array}
$$



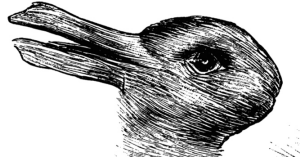
If-Then ambiguity, aka the dangling-else problem



$$
\begin{array}{rcll}
M & \rightarrow & \mathit{OPEN_IF} & \text{// some 'if's have no 'else's} \\
 & | & \mathit{CL_IF} & \text{// only 'if's with 'else's} \\
 & & & \\
\mathit{OPEN_IF} & \rightarrow & \texttt{if}\ E\ \texttt{then}\ M & \text{// 'if' without matching 'else'} \\
 & | & \texttt{if}\ E\ \texttt{then}\ \mathit{CL_IF}\ \texttt{else}\ \mathit{OPEN_IF} & \text{// 'inner if's are closed} \\
 & & & \\
\mathit{CL_IF} & \rightarrow & \texttt{if}\ E\ \texttt{then}\ \mathit{CL_IF}\ \texttt{else}\ \mathit{CL_IF} & \text{// nicely closed 'if's} \\
 & | & \mathit{NON_IF_STMTS} & \text{// no 'if's at all}
\end{array}
$$

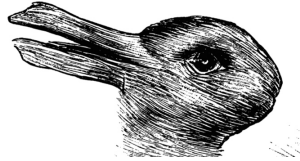


Ambiguity: general algorithm?



There do exist some inherently ambiguous languages
— so, there is no way to take an ambiguous grammar,
and produce an unambiguous grammar

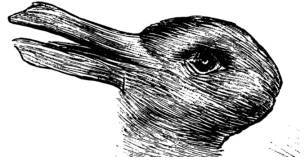
Ambiguity: general algorithm?



There do exist some inherently ambiguous languages
— so, there is no way to take an ambiguous grammar,
and produce an unambiguous grammar

Fortunately, such languages are not generally relevant for programming languages: we can usually take an ambiguous grammar, and make it unambiguous.

Ambiguity: general algorithm?



There do exist some inherently ambiguous languages
— so, there is no way to take an ambiguous grammar,
and produce an unambiguous grammar

Fortunately, such languages are not generally relevant for programming languages: we can usually take an ambiguous grammar, and make it unambiguous.

I will not ask you in the exam to convert an ambiguous CFG into an unambiguous CFG — but it is important that you know about ambiguity, and that it is important to address

Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.

Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

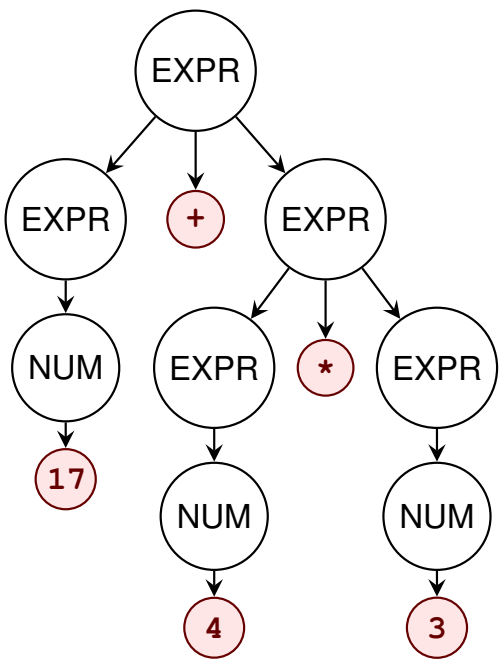
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



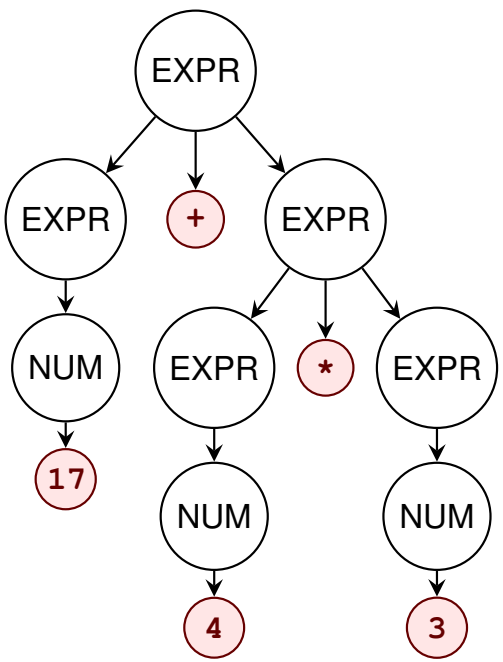
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR

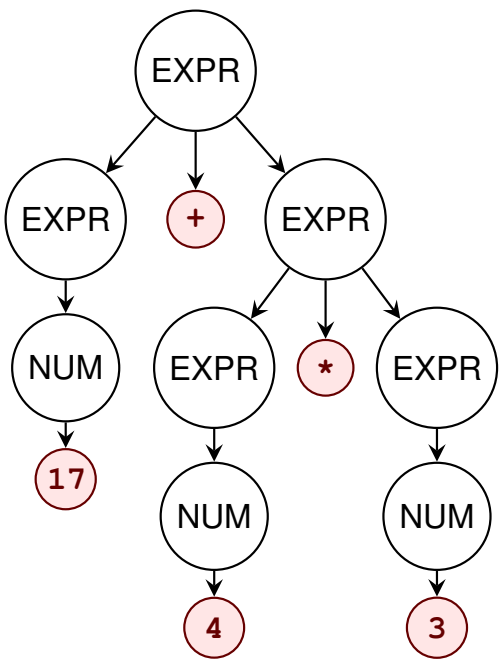
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR

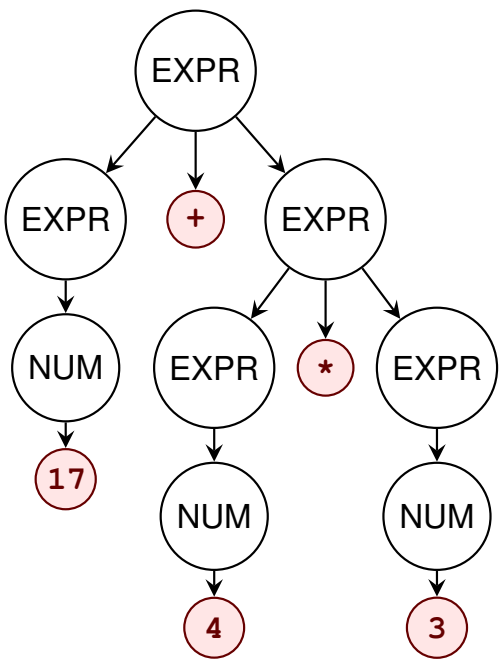
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR

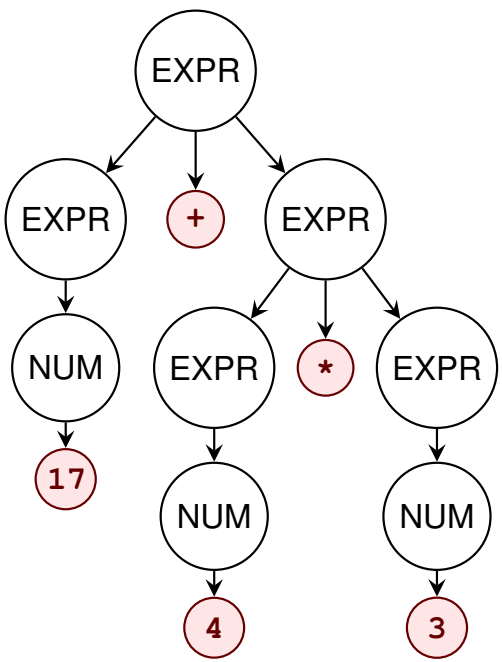
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR

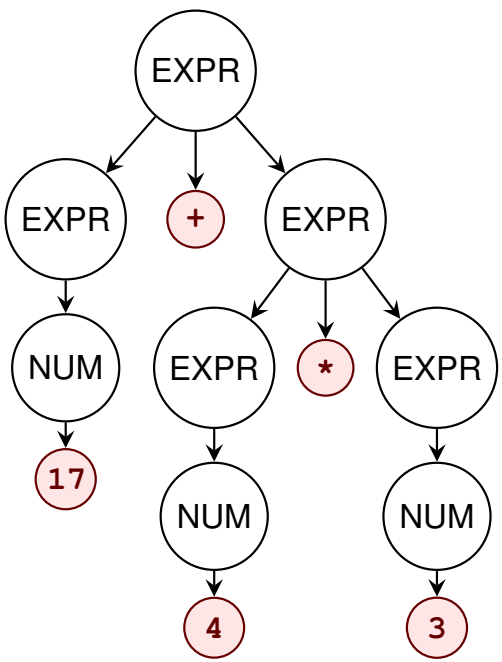
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR

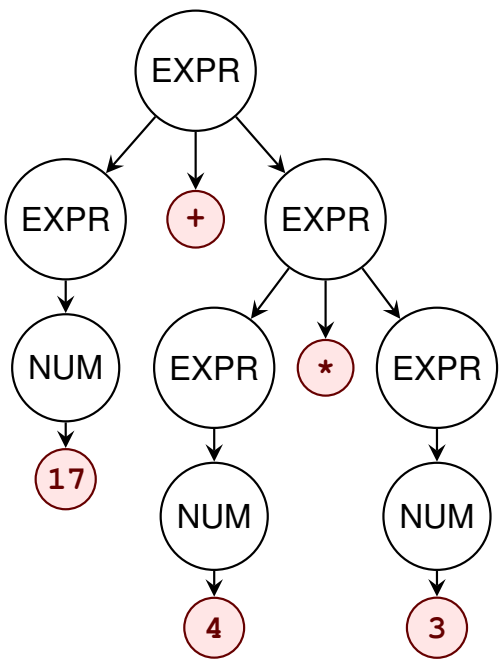
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR
`17 +` NUM `*` EXPR

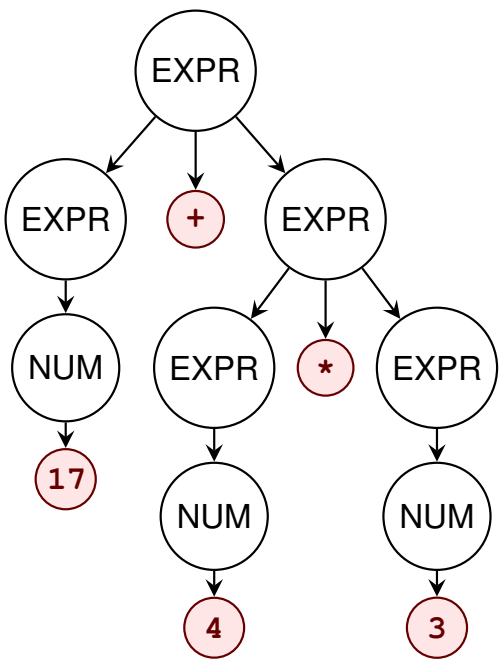
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR
`17 +` NUM `*` EXPR
`17 + 4 *` EXPR

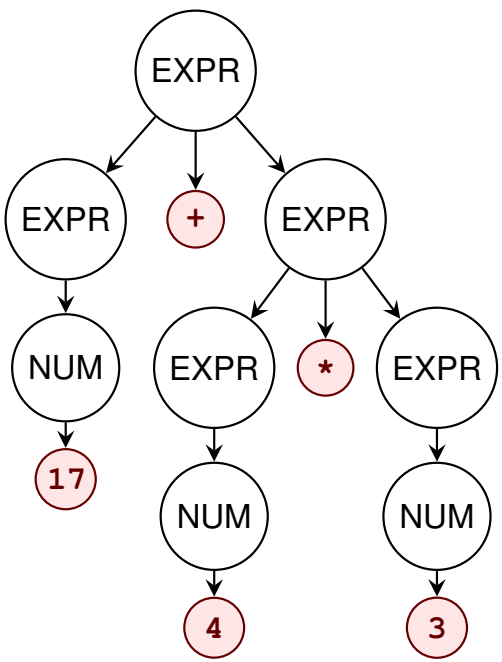
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR
`17 +` NUM `*` EXPR
`17 + 4 *` EXPR
`17 + 4 *` NUM

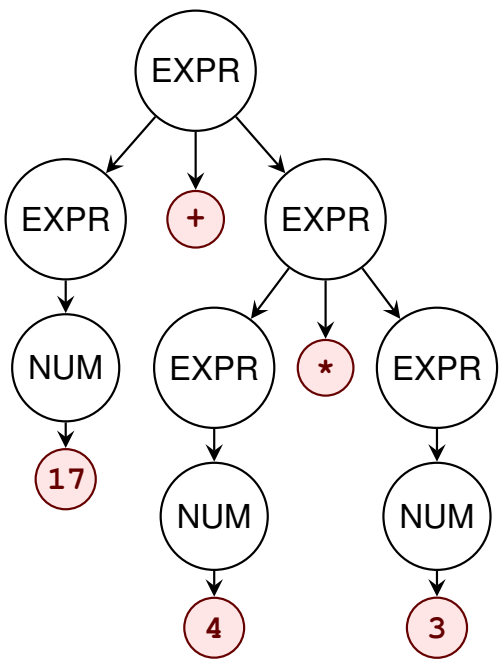
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR
`17 +` NUM `*` EXPR
`17 + 4 *` EXPR
`17 + 4 *` NUM
`17 + 4 * 3`

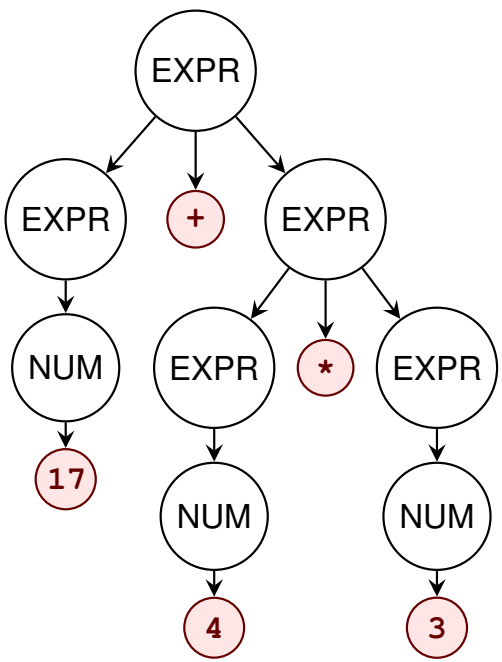
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR

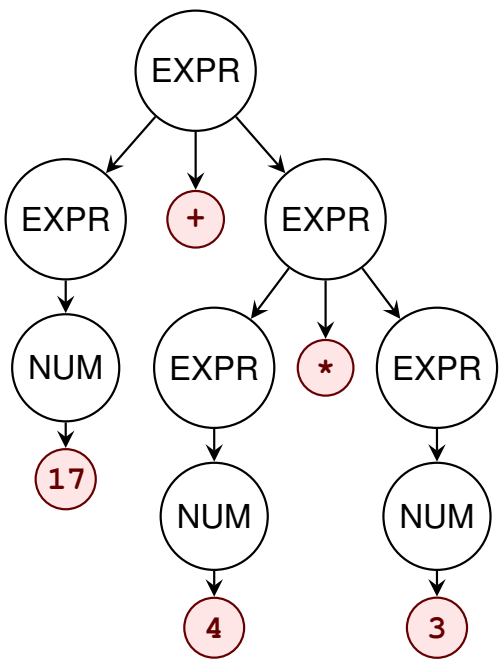
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR

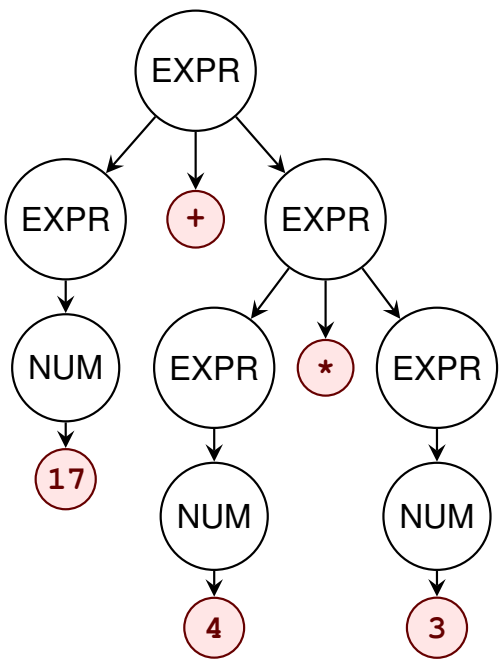
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR
EXPR + EXPR * EXPR

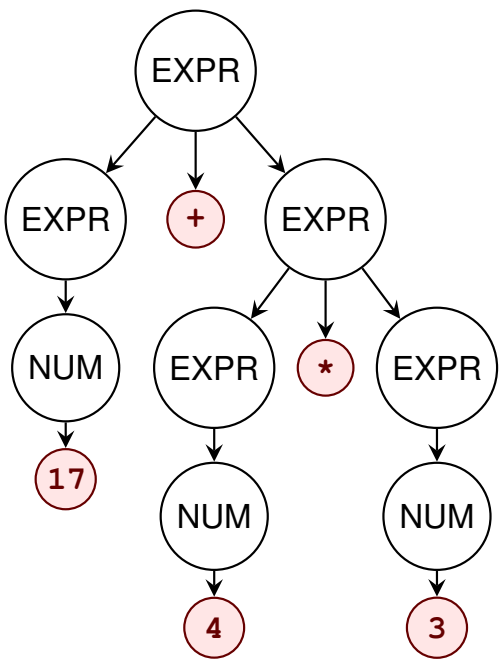
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR
EXPR + EXPR * EXPR
EXPR + EXPR * NUM

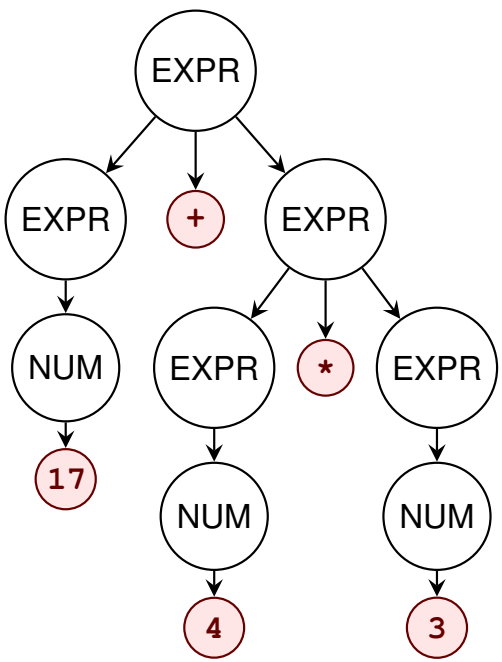
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR
EXPR + EXPR * EXPR
EXPR + EXPR * NUM
EXPR + EXPR * 3

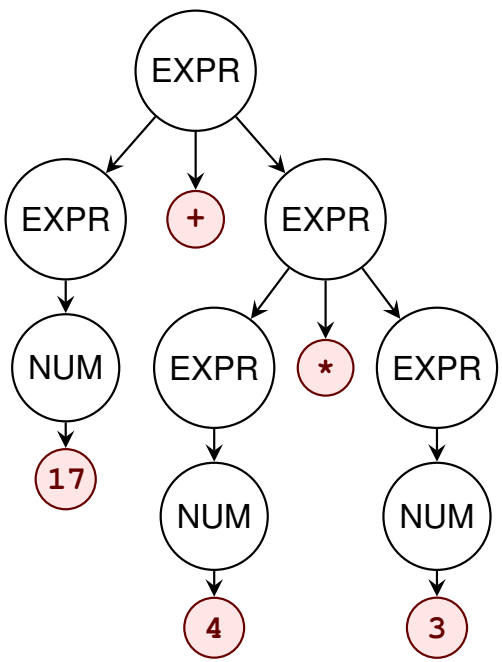
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR `+` EXPR
NUM `+` EXPR
`17 +` EXPR
`17 +` EXPR `*` EXPR
`17 +` NUM `*` EXPR
`17 + 4 *` EXPR
`17 + 4 *` NUM
`17 + 4 * 3`

EXPR
EXPR `+` EXPR
EXPR `+` EXPR `*` EXPR
EXPR `+` EXPR `*` NUM
EXPR `+` EXPR `* 3`
EXPR `+` NUM `* 3`

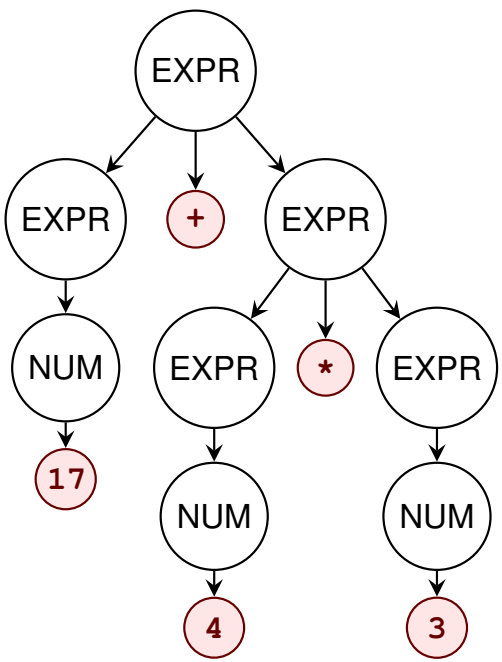
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR
EXPR + EXPR * EXPR
EXPR + EXPR * NUM
EXPR + EXPR * 3
EXPR + NUM * 3
EXPR + 4 * 3

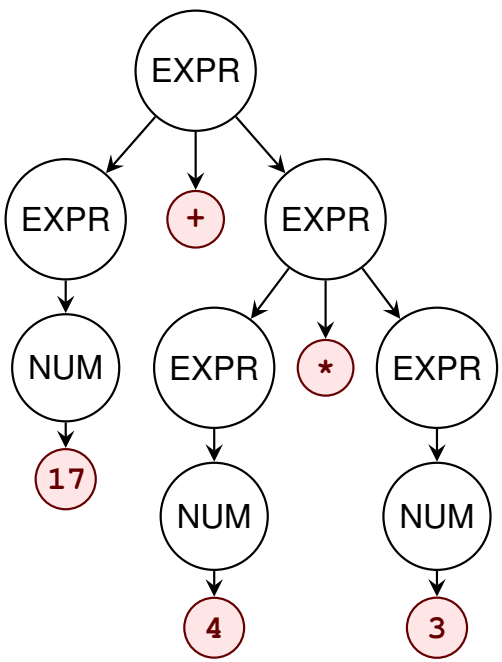
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR
EXPR + EXPR
NUM + EXPR
17 + EXPR
17 + EXPR * EXPR
17 + NUM * EXPR
17 + 4 * EXPR
17 + 4 * NUM
17 + 4 * 3

EXPR
EXPR + EXPR
EXPR + EXPR * EXPR
EXPR + EXPR * NUM
EXPR + EXPR * 3
EXPR + NUM * 3
EXPR + 4 * 3
NUM + 4 * 3

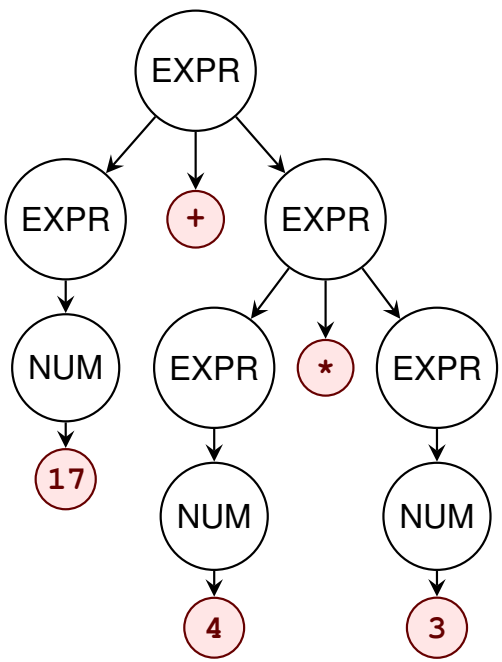
Left- vs rightmost derivation

There are usually many **derivations** for a chosen string (even if they all produce the same parse tree)

Canonical choices (there are usually may others!):

- *Left-most:* always replace the left-most variable.
- *Right-most:* always replace the right-most variable.

Example.



EXPR	EXPR
EXPR + EXPR	EXPR + EXPR
NUM + EXPR	EXPR + EXPR * EXPR
17 + EXPR	EXPR + EXPR * NUM
17 + EXPR * EXPR	EXPR + EXPR * 3
17 + NUM * EXPR	EXPR + NUM * 3
17 + 4 * EXPR	EXPR + 4 * 3
17 + 4 * NUM	NUM + 4 * 3
17 + 4 * 3	17 + 4 * 3

Left- vs rightmost derivation

For an unambiguous CFG:
each input string has only one parse tree.

Left- vs rightmost derivation

For an unambiguous CFG:
each input string has only one parse tree.

But: for any actually interesting CFG and input string,
the left-most and right-most derivations will be **different**.

Left- vs rightmost derivation

For an unambiguous CFG:
each input string has only one parse tree.

But: for any actually interesting CFG and input string,
the left-most and right-most derivations will be **different**.

Left-most and right-most derivations correspond to ways
that we try to solve the problem of **constructing the AST** —
that is: of parsing.