If an Infix Expression e Has 2 or More Operators, Which of Those Operators Should be Applied Last?

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a unary operator, *unless* **op** *is at the very beginning or at the very end of* e.

Example: The top-level operators in the C++ expression

`- x / - (y + (z - 3) - 2) * w ++ % v ++`

are the *1st* unary -, /, *, %, and the *2nd* ++.

The three operators in `(y + (z - 3) - 2)` are ***not*** top-level: They violate (a).

The 2nd unary - is ***not*** top-level: It violates (b).

The *1st* ++ is ***not*** top-level: It violates (b).

An example of precedence and associativity rules (from the course reader).

assignment	=
logical or	||
logical and	&&
inclusive or	|
exclusive or	^
and	&
equality	== !=
relational	< <= >= >
shift	<< >>
additive	+ -
multiplicative	* / %

Figure 2.9 A partial table of binary operators in C, in order of increasing precedence; that is, the assignment operator = has the lowest precedence and the multiplicative operators *, /, and % have the highest precedence. All operators on the same line have the same precedence and associativity. The assignment operator is right associative; all the other operators are left associative.

If an Infix Expression e Has 2 or More Operators, Which of Those Operators Should be Applied Last?

If no precedence and associativity rules for the operators are given, then when an expression e has two or more top-level operators it is ***not*** possible to uniquely determine which operator of e should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i) partition the operators into ***ranked precedence classes,*** and

(ii) specify, for each class, whether the class is ***left***-associative (= ***left-to-right*** associative) or ***right***-associative (= ***right-to-left*** associative).

Using these rules, ***we can find the operator of e that should be applied last as follows***:

If an infix expression e has 2 or more operators, you can find the operator that is applied last as follows:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.
2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: `x * (y + (z + 3) - 2) + w - u / t`

Solution: There are 4 top-level operators, namely the 4 **black** operators in: `x * (y + (z + 3) - 2) + w - u / t`

Step 2.1: The top-level operators of ***lowest precedence*** are the 2 blue operators in: `x * (y + (z + 3) - 2) + w - u / t`

Step 2.3: These 2 operators + and - belong to a ***left-associative*** precedence class, so the rightmost of them (i.e., -) must be applied last.

Precedence & Associativity Rules Might ***Not*** Uniquely Determine Which Operator Should be Applied ***First***

Consider this C/C++ expression: **`y / 2 * --y`**

Here the top-level operators (**/** and *****) lie in a ***left-assoc.*** prec. class: So ***** should be applied ***last***.

But a C or C++ compiler is free to generate code that applies **--** first or applies **/** first!

Note: In addition to precedence & associativity rules, a language may have *other* rules that govern the order in which operators in infix expressions are applied!

Example: In **Java**, arguments of functions or operators are evaluated in ***left-to-right*** order, so **/** is applied ***before*** **--** when evaluating the above expression. Thus

```
int y = 4; System.out.println(y / 2 * --y);
```

prints 6 (not 3). In C++ there's no left-to-right rule, so **`int y = 4; cout << y / 2 * --y;`** may print 3 or 6.

Example

In a certain language expressions are written in infix notation. The operators that may appear in expressions fall into four precedence classes, which are specified in the table below. For $1 \leq i < 4$, class i has ***higher*** precedence than class i+1 (so class 1 has highest precedence).

	prefix unary ops	binary ops	associativity
Class 1	~		*right*-associative
Class 2	+ –	+ –	*left*-associative
Class 3		& ^ @	*right*-associative
Class 4		# $	*left*-associative

Circle the operator that should be applied ***last*** when evaluating the following expression:

+ x **@** (z **&** ~ y ^ z) **&** (a **@** ~ z ^ x) **&** y **–** 1

RECALL: We can find the operator that is applied last in e as follows:

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

The five **black** operators are the ***top-level*** operators, and so there are three ***top-level*** operators of ***lowest*** precedence: the **@** and the two **&**s.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found if their precedence class is ***right***-associative.

The three operators found by 2.1 are in a ***right****-associative class*, so the ***leftmost*** of those three operators (i.e., **@**) should be applied last.

Another Example

In a certain language expressions are written in infix notation. The operators that may appear in expressions fall into four precedence classes, which are specified in the table below. For $1 \leq i < 4$, class i has ***higher*** precedence than class i+1 (so class 1 has highest precedence).

	prefix unary ops	binary ops	associativity
Class 1	~		*right*-associative
Class 2	+ –	+ –	*left*-associative
Class 3		& ^ @	*right*-associative
Class 4		# $	*left*-associative

Circle the operator that should be applied ***last*** when evaluating the following expression:

– x **+** z **$** (~ y ^ z) **&** a **@** ~ z ^ x **#** y **–** 1

RECALL: We can find the operator that is applied last in e as follows:

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

The eight **black** operators are the ***top-level*** operators, and so there are two ***top-level*** operators of ***lowest*** precedence: the **$** and the **#**.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found if their precedence class is ***right***-associative.

The two operators found by 2.1 are in a ***left****-associative class*, so the ***rightmost*** of those two operators (i.e., **#**) should be applied last.

More on Postfix & Prefix Notations

Syntactically Valid Prefix & Postfix Expressions

Unlike infix notations, postfix and prefix notations allow operators of arity *k* for any positive integer *k*.

An expression e is said to be a ***syntactically valid prefix expression*** (***s.v.pre.e.***) if one of the following is true:

1. e is an identifier or a literal constant.
2. e = **op** e_1 e_2 ... e_k where **op** is a *k*-ary operator and each of e_1, e_2, ..., e_k is an s.v.pre.e.

An expression e is said to be a ***syntactically valid postfix expression*** (***s.v.post.e.***) if one of the following is true:

1. e is an identifier or a literal constant.
2. e = e_1 e_2 ... e_k **op** where **op** is a *k*-ary operator and each of e_1, e_2, ..., e_k is an s.v.post.e.

Semantics of a Prefix Expression e

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant, then
   e.value = the value of the identifier / constant.
2. If e = **op** e_1 e_2 ... e_k where **op** is a k-ary operator and each of e_1, e_2, ..., e_k is a **prefix** expression, then
   e.value = the result of applying **op** with e_i.value as its i^{th} argument ($1 \leq i \leq k$).

Semantics of a Postfix Expression e

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant, then
   e.value = the value of the identifier / constant.
2. If e = e_1 e_2 ... e_k **op** where **op** is a k-ary operator and each of e_1, e_2, ..., e_k is a **postfix** expression, then
   e.value = the result of applying **op** with e_i.value as its i^{th} argument ($1 \leq i \leq k$).

Some Notable Differences Between Prefix/Postfix and Infix Notations

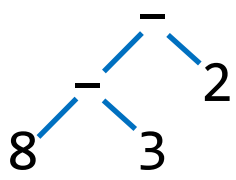
- Prefix and postfix notations are parenthesis-free.
- Operators of arity > 2 are allowed in prefix and postfix notations, but not in infix notation.

  However, a prefix or postfix expression may be ambiguous if you don't know the arities of operators.
- In prefix and postfix notations, operators are ***not*** divided into different precedence classes.
- In prefix and postfix notations, there is no concept of left- or right-associativity.

Abstract Syntax Trees

Even though it is called a "syntax tree" the **abstract syntax tree** (AST) of an expression is a tree that represents an expression's *semantics* (meaning).

For example, the Lisp expression (– (– 8 3) 2) and the two Java expressions 8 – 3 – 2 and ((8 – 3) – 2) all have the following AST:



- Two expressions are equivalent (i.e., have the same semantics) ***if and only if*** they have the same AST.

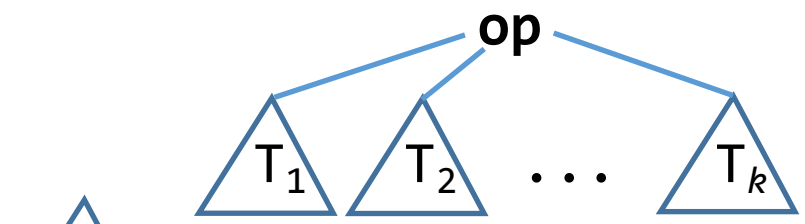
Thus the above three expressions are equivalent.

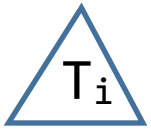
Note that ASTs do ***not*** have parentheses as nodes!

The ***abstract syntax tree*** (AST) of an expression e can be defined as follows:

1. If e contains ***no*** operator, then e is equivalent to a variable or constant. In this case e's AST *has just one node*, which is the variable or constant itself.

2. In all other cases, let **op** be the operator of e that should be applied *last* when evaluating e, let *k* be the arity of **op**, and let e_1, ..., e_k be the subexpressions that are the *k* operands of **op** (where e_i is the *i*th operand). Then the AST of e is



where 

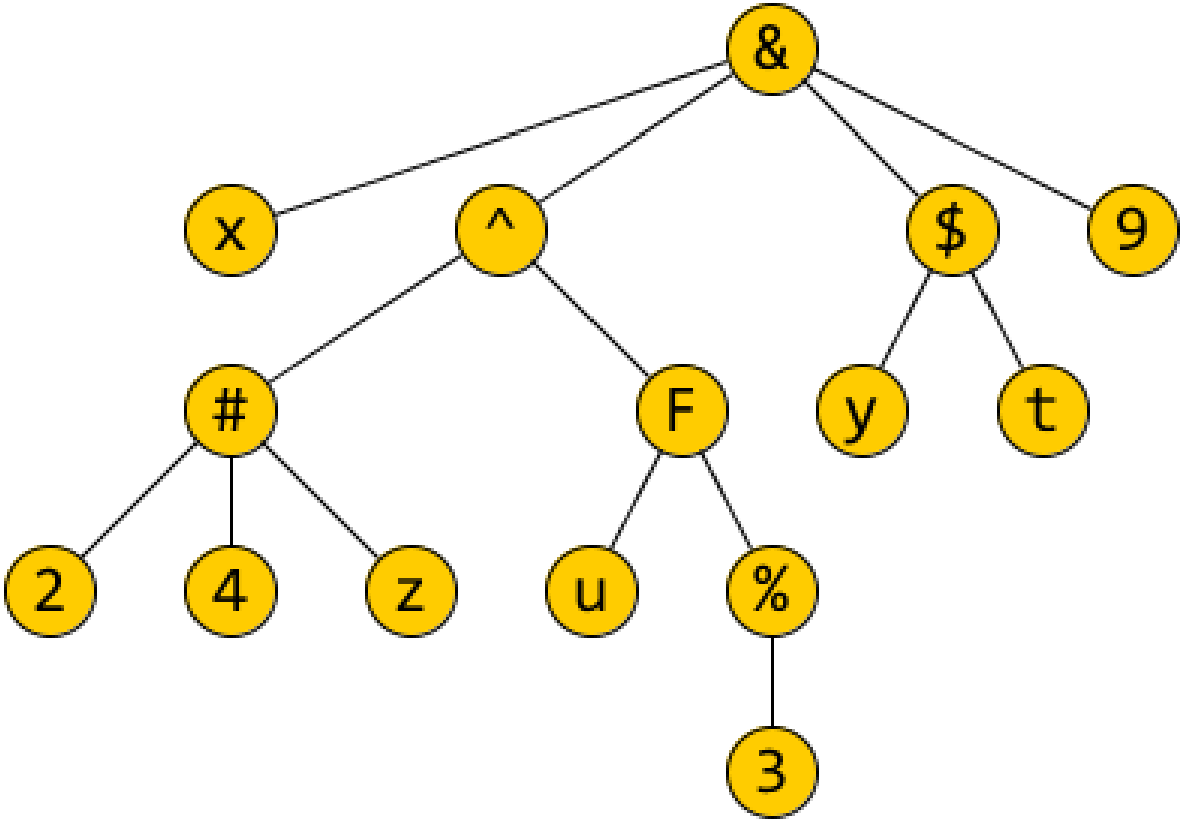
is the AST of the subexpression e_i.

ASTs of *infix* expressions are binary trees, because infix notation doesn't allow operators of arity > 2.

Example: Draw the AST of the following expression, which is written in Lisp notation:
(& x (^ (# 2 4 z) (F u (% 3))) ($ y t) 9)

Solution:

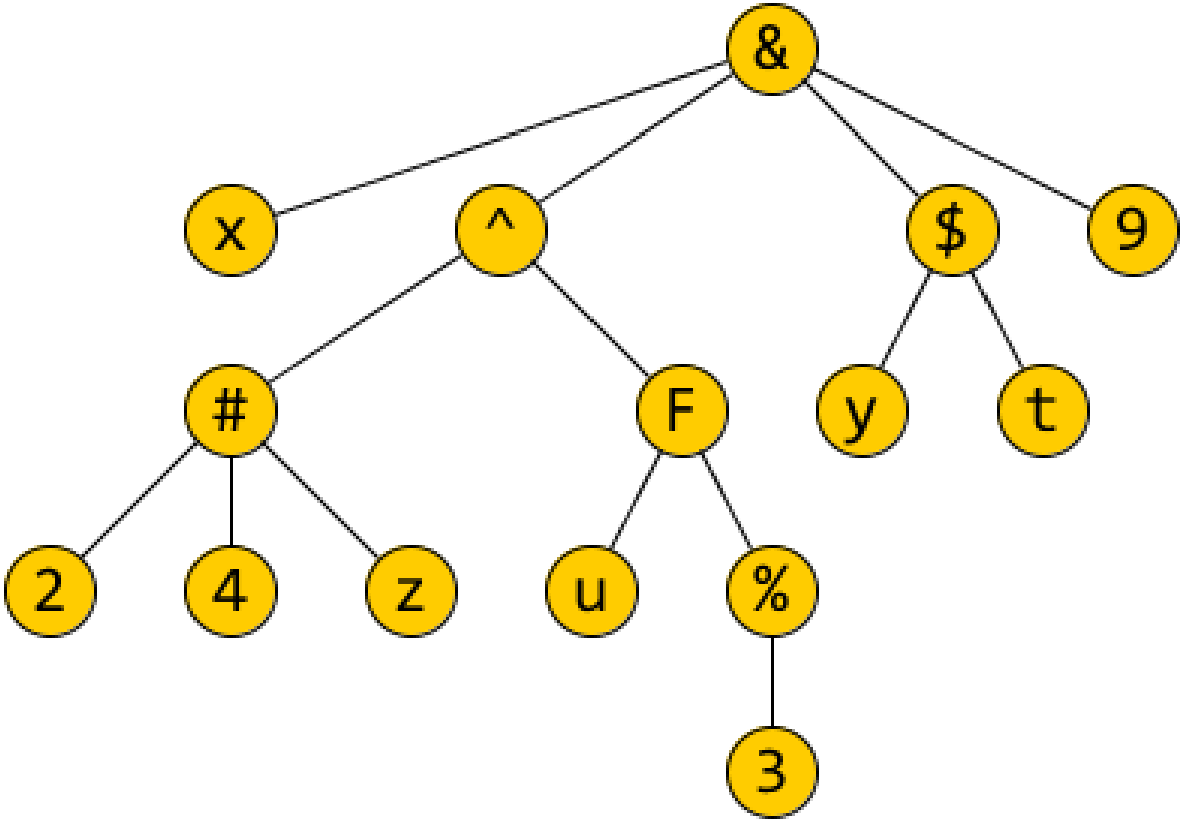
This is the AST of:
(& x (^ (# 2 4 z) (F u (% 3))) ($ y t) 9)



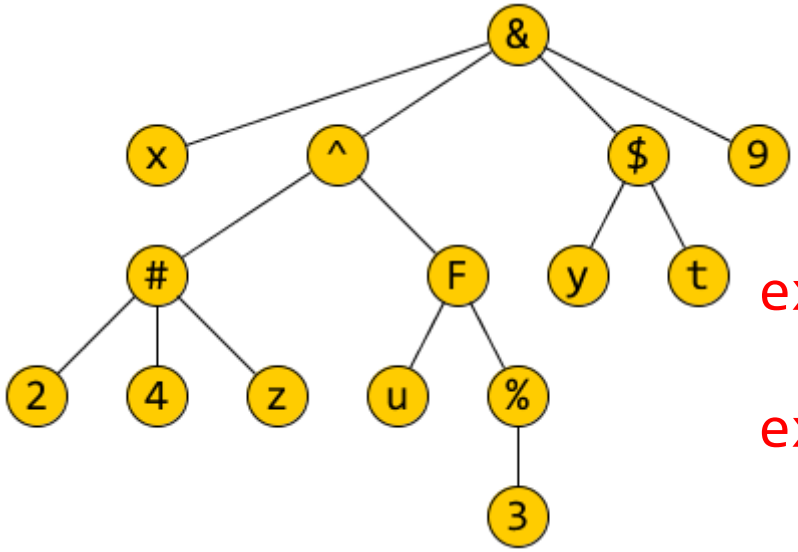
Example: Draw the AST of the following expression, which is written in "rpnLisp" notation:
(x ((2 4 z #) (u (3 %) F) ^) (y t $) 9 &)

Solution:

This is the AST of:
(x ((2 4 z #) (u (3 %) F) ^) (y t $) 9 &)



- To draw a prefix expression's AST, you can write an equivalent Lisp expression and then draw its AST.
- To draw a postfix expression's AST, you can write an equivalent rpnLisp expression and then draw its AST.
- ***Preorder*** traversal of an expression's AST will give an equivalent expression in ***prefix*** notation.
- ***Postorder*** traversal of an expression's AST will give an equivalent expression in ***postfix*** notation.



expression in ***prefix*** notation:

& x ^ # 2 4 z F u % 3 $ y t 9

expression in ***postfix*** notation:

x 2 4 z # u 3 % F ^ y t $ 9 &

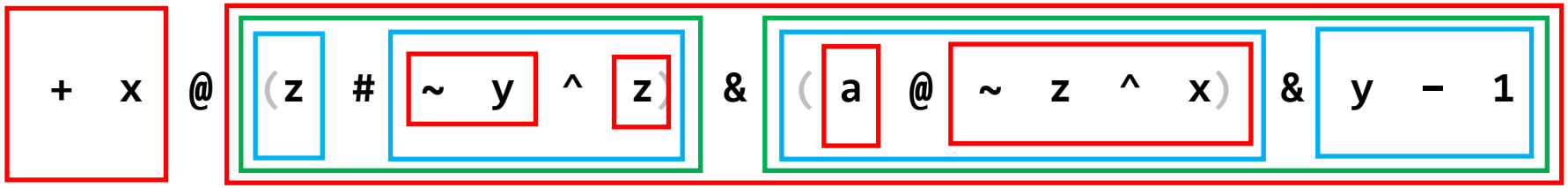
Example: Draw the AST of the infix expression

+ x @ (z # ~ y ^ z) & (a @ ~ z ^ x) & y - 1

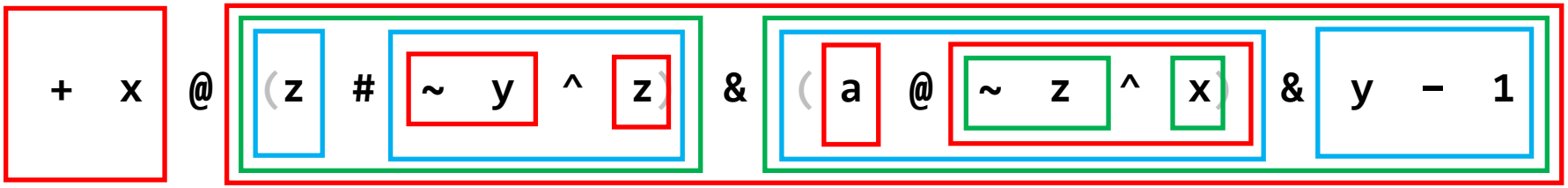
assuming the operators' precedence classses are as follows:

	prefix unary ops	binary ops	associativity
Class 1	~		*right*-associative
Class 2	+ -	+ -	*left*-associative
Class 3		& ^ @	*right*-associative
Class 4		# $	*left*-associative

For $1 \leq i < 4$, class i has ***higher*** precedence than class i+1.

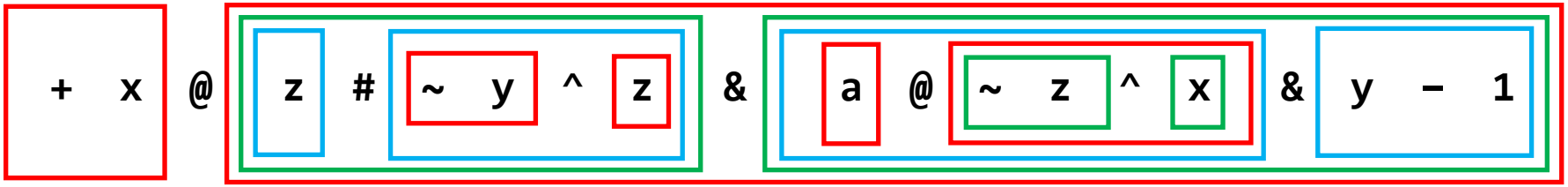


One of the subexpressions in the inner red boxes has more than one operator. Find the operator of that subexpression that's applied last, and its operands:

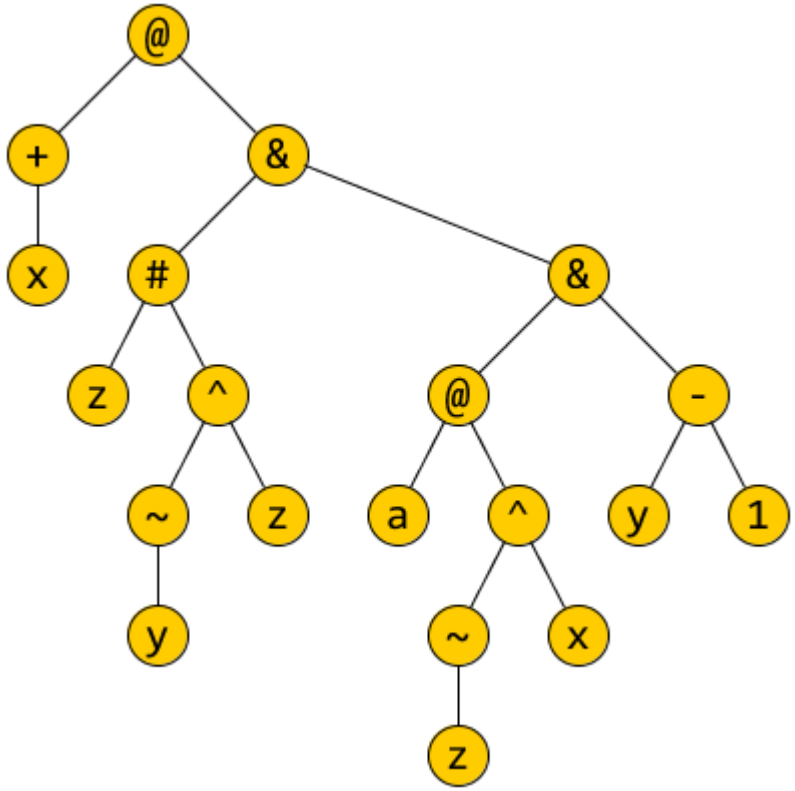


Example: Draw the AST of the infix expression

+ x @ (z # ~ y ^ z) & (a @ ~ z ^ x) & y – 1



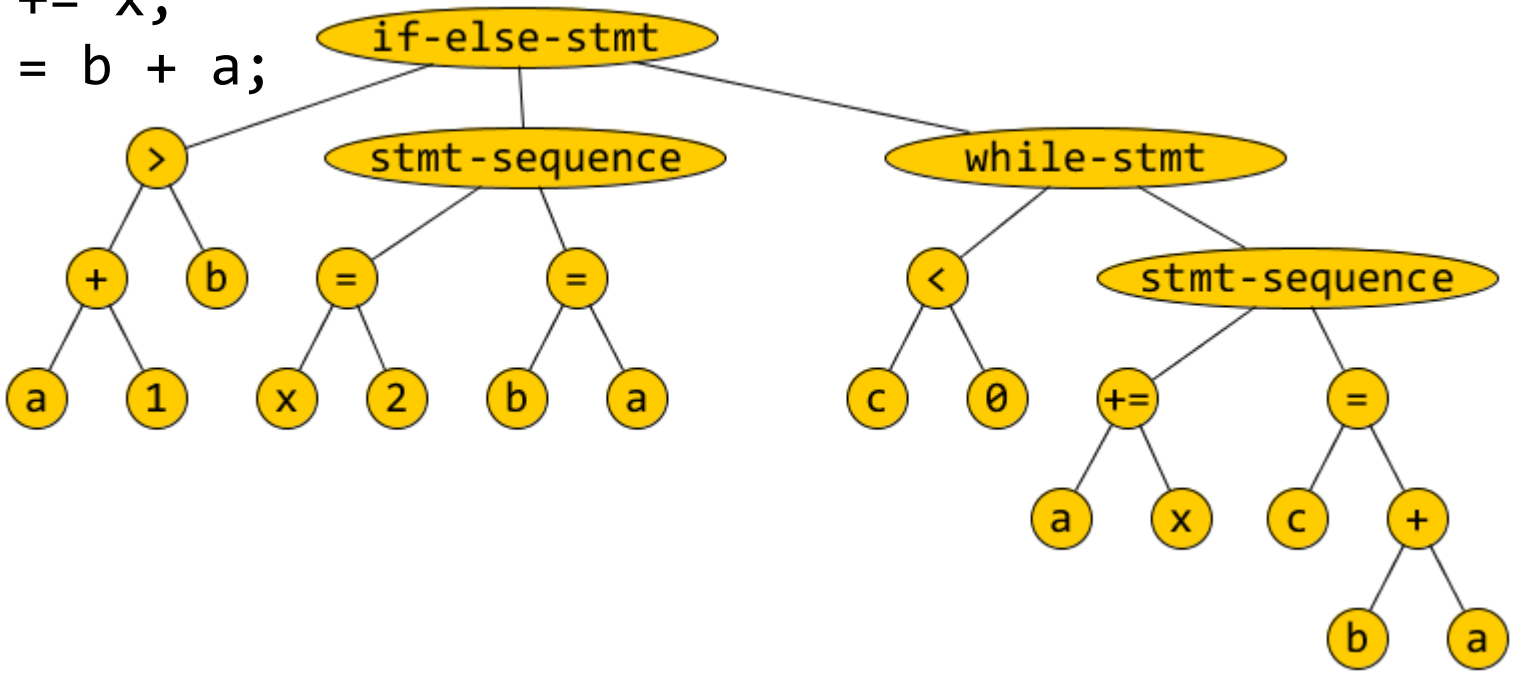
As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:



Example of a Possible AST of a Java Statement

```
if (a+1 > b) {
  x = 2;
  b = a;
}
else {
  while (c < 0) {
    a += x;
    c = b + a;
  }
}
```

can be represented by the following AST:



Evaluation of Postfix Expressions Using a Stack

Postfix expressions can be evaluated as follows:

- Read the expression *from left to right*.
- When a variable or constant is seen, *push* its value.
- When a *k*-ary operator **op** is seen, *pop* off *k* values, *apply* **op** to those values (with the i^{th}-last value to be popped as the i^{th} argument), and *push* the result.

After the entire expression has been processed in this way, the value of the expression will be the only thing on the stack.

The last homework exercise in section A of https://euclid.cs.qc.cuny.edu/316/Syntax-Reading-and-Exercises.pdf asks you to evaluate a postfix expression in this way!

Prefix expressions can be evaluated in a similar way, if we read the expression from ***right to left***.

Evaluation of Postfix Expressions Using a Stack

Postfix expressions can be evaluated as follows:

- Read the expression *from left to right*.
- When a variable or constant is seen, *push* its value.
- When a k-ary operator **op** is seen, *pop* off k values, *apply* **op** to those values (with the i^{th}-last value to be popped as the i^{th} argument), and *push* the result.

After the entire expression has been processed in this way, the value of the expression will be the only thing on the stack.

Example Let $+_3$ and $*_3$ be the 3-ary plus and times operators, and let $*_2$ and $-_2$ the binary times and minus operators.
Suppose that x has value 7 and that y has value 4.
We now show evaluation of: **x 2 3 y $+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

value of expression

***UNREAD* INPUT:**

STACK (**rightmost** item = topmost item)**:** **70**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

- Read the expression *from right to left*.
- When a variable or constant is seen, *push* its value.
- When a k-ary operator **op** is seen, *pop* off k values, *apply* **op** to those values (with the i^{th} value to be popped as the i^{th} argument), and *push* the result.

After the entire expression has been processed in this way, the value of the expression will be the only thing on the stack.

Example Let $+_3$ and $*_3$ be the 3-ary plus and times operators, and let $*_2$ and $-_2$ the binary times and minus operators.
Suppose that x has value 7 and that y has value 4.
We now show evaluation of: $*_3$ **x** $-_2$ $+_3$ **2 3 y** $*_2$ **1 x 5**

***UNREAD* INPUT:**

STACK (leftmost item = topmost item): 70

value of expression