## More on Infix Notation

Infix notation allows unary and binary operators, but does ***not*** allow operators of arity > 2.

*Binary* operators are written between their operands.

The designer of an infix notation must specify, for each *unary* operator the notation allows, whether that unary operator is to be written as a ***prefix operator*** or is to be written as a ***postfix operator***:

- Prefix operators are written *before* their operands.
  - **Examples**: `-` in a Java or C++ expression `-x`
    `++` in a Java or C++ expression `++i`
    `*` in a C++ expression `*ptr`
- Postfix operators are written *after* their operands.
  - **Examples**: `++` in a Java or C++ expression `i++`
    `[e]` acts like a postfix operator in `a[e]`.

## Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = ( $e_1$ ), where $e_1$ is an s.v.i.e.
3. e = $e_1$ **op** $e_2$ where each of $e_1$ and $e_2$ is an s.v.i.e. and **op** is a binary operator.
4. e = **op** $e_1$ where $e_1$ is an s.v.i.e. and **op** is a prefix unary operator.
5. e = $e_1$ **op** where $e_1$ is an s.v.i.e. and **op** is a postfix unary operator.

Rules 3 – 5 give decompositions of e into two (rules 4 & 5) or three (rule 3) substructures, but some of these decompositions may ***violate*** the following important principle of syntax specification:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

## Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be this Java expression: `– x – y * z + w`

Then 3. e = $e_1$ **op** $e_2$ where each of $e_1$ and $e_2$ is an s.v.i.e. and **op** is a binary operator.

and 4. e = **op** $e_1$ where $e_1$ is an s.v.i.e. and **op** is a prefix unary operator.

give the following decompositions of e:

| | | | |
|---|---|---|---|
| (i) | $e_1$ = `– x` | **op** = `–` | $e_2$ = `y * z + w` |
| (ii) | $e_1$ = `– x – y` | **op** = `*` | $e_2$ = `z + w` |
| (iii) | $e_1$ = `– x – y * z` | **op** = `+` | $e_2$ = `w` |
| (iv) | **op** = `–` | $e_1$ = `x – y * z + w` | |

Decompositions (i), (ii), and (iv) ***violate*** the principle, as Java's semantics say e is equivalent to: `(– x – y * z) + w`

- Sec. 2.5 of Sethi (assigned reading after Exam 1) gives another way to specify syntactically valid infix expressions *that* ***doesn't allow bad decompositions like*** (i), (ii), and (iv).

**Semantics of an Infix Expression e**

The semantics of e tells you how e can be evaluated.

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

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.

2. If e = ($e_1$), e.value = $e_1$.value.

Otherwise, ***let*** **op** ***be the operator of*** e ***that is applied last***: **Rule T** below will imply e is $e_1$ **op** $e_2$, **op** $e_1$, or $e_1$ **op**, where $e_1$ is an s.v.i.e and so is $e_2$ in the first case.

- If e is $e_1$ **op** $e_2$,
  e.value = result of applying **op** with $e_1$.value and $e_2$.value as the 1st and 2nd arguments.
- If e is **op** $e_1$ or e is $e_1$ **op**,
  e.value = result of applying **op** to $e_1$.value.

**Key Question**: How can we determine which operator of e should be applied **last**?

**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) ***either*** **op** is a binary operator in e
***or*** **op** is a unary operator *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*** **++**.

**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) ***either*** **op** is a binary operator in e
***or*** **op** is a unary operator *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).

**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) ***either*** **op** is a binary operator in e
***or*** **op** is a unary operator *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).

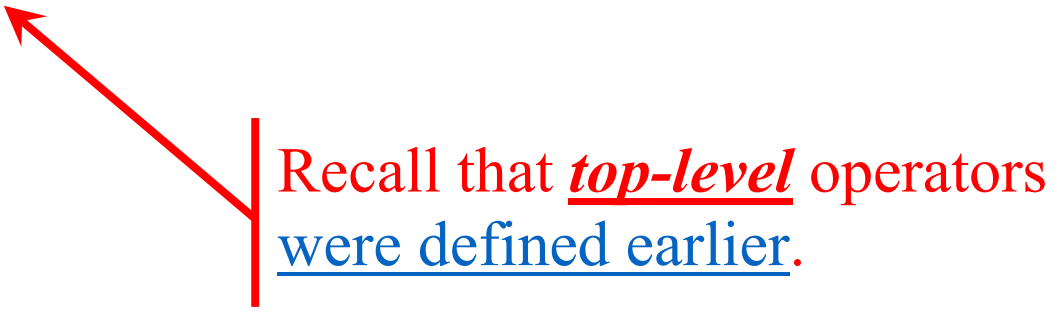
The 2nd unary `-` and first unary `++` are ***not*** top-level: They violate (b).

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

If e is of the form (e′), then the operator that is applied last in e′ is also the operator that is applied last in e.

Otherwise, the following rule applies:

**Rule T:** The operator of e that is applied last ***must be a top-level operator of*** e.



**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 (unless some other rule or rules are imposed)!

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).

**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 (unless some other rule or rules are imposed)!

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***:

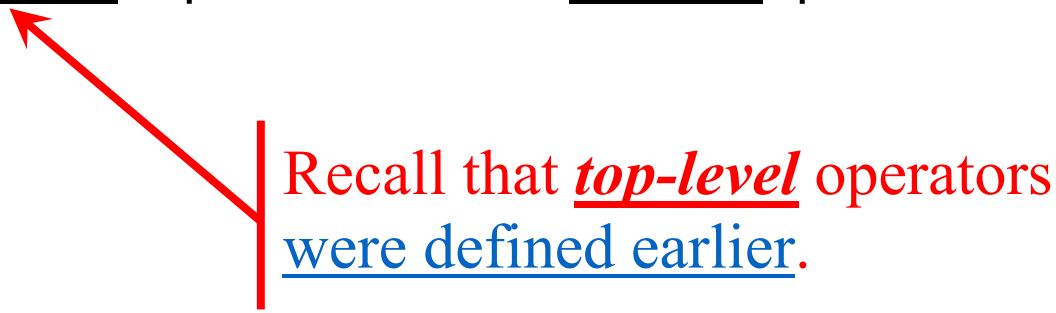
**If an infix expression e has 2 or more operators, you can find the operator that is applied last as follows:**

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

2.3



**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 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.

**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:

Step 2.3:

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

From the course reader.

**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.

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.

**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.

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 if their precedence class is ***right***-associative.

**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.

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 if their precedence class is ***right***-associative.

**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:

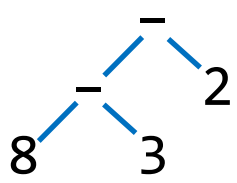
1. If e is an identifier or a literal constant, then
   e.value = the value of the identifier / constant.
2. If e = $e_1$ $e_2$ ... $e_k$ **op** where **op** is a $k$-ary operator and each of $e_1$, $e_2$, ..., $e_k$ is a **postfix** expression, then
   e.value = the result of applying **op** with $e_i$.value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

## 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).

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

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



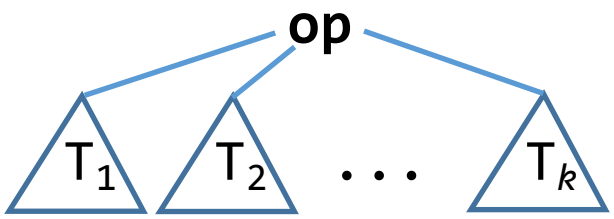

*Important properties of ASTs include*:

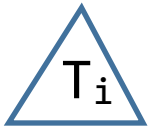
- Each AST node is *a variable or a constant* or is *an operator*; ASTs do ***not*** have parentheses as nodes!
- A *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.

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 $e_i$.

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

**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.

**Solution:** First we find the operator that's applied last, and the operands of that operator:

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

In the subexpression in the 2nd red box, find the operator that's applied last and its operands:

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

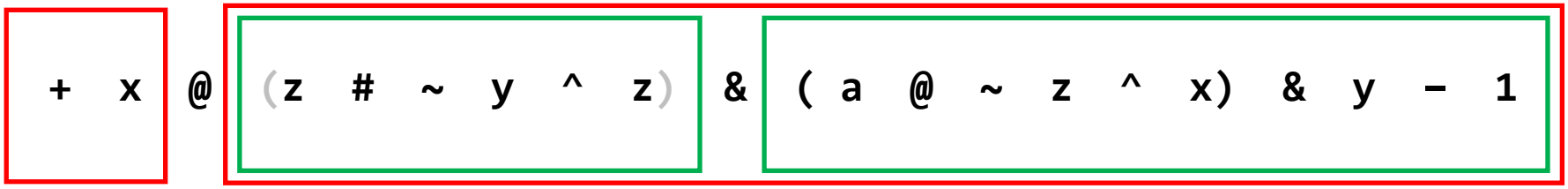
**Example:** Draw the AST of the infix expression

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

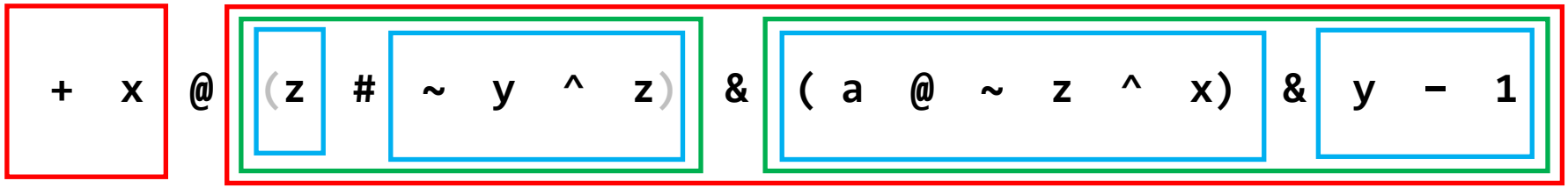
assuming the operators' precedence classses are as follows:

| | 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.




The two subexpressions in green boxes each have more than one operator. In each case, find the operator that's applied last and its operands:



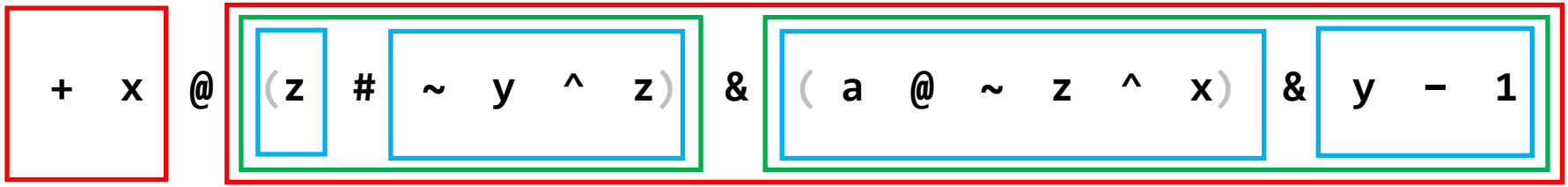
**Example:** Draw the AST of the infix expression

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

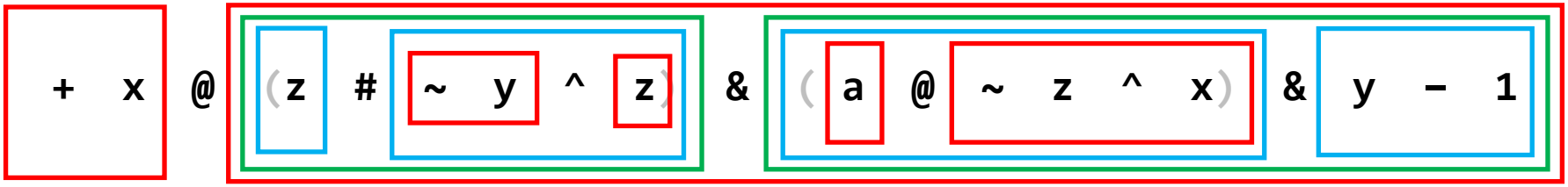
assuming the operators' precedence classses are as follows:

| | 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.




Two of the subexpressions in blue boxes have more than one operator. In each case, find the operator that's applied last and its operands:



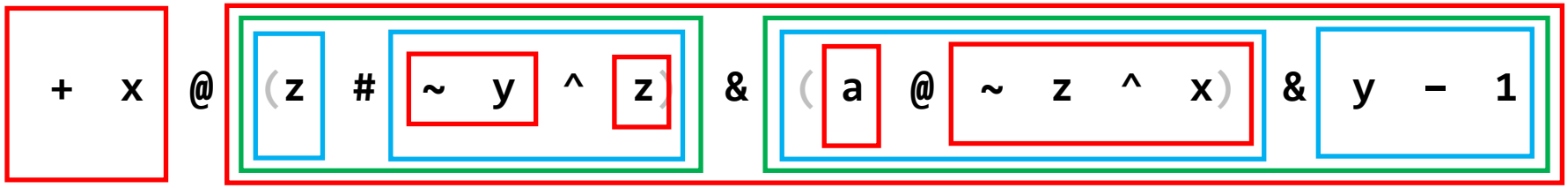
**Example:** Draw the AST of the infix expression

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

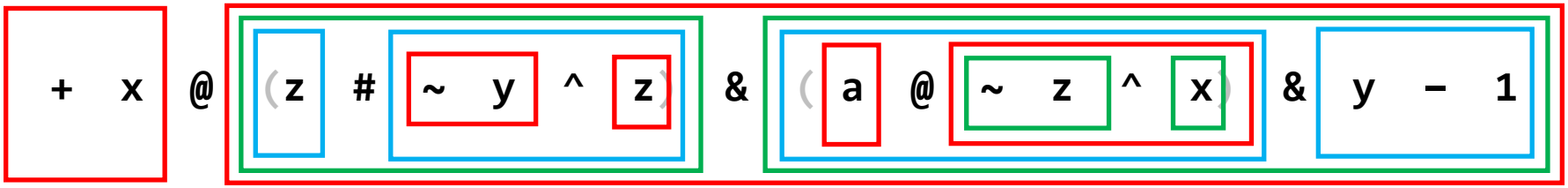
assuming the operators' precedence classses are as follows:

| | 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.

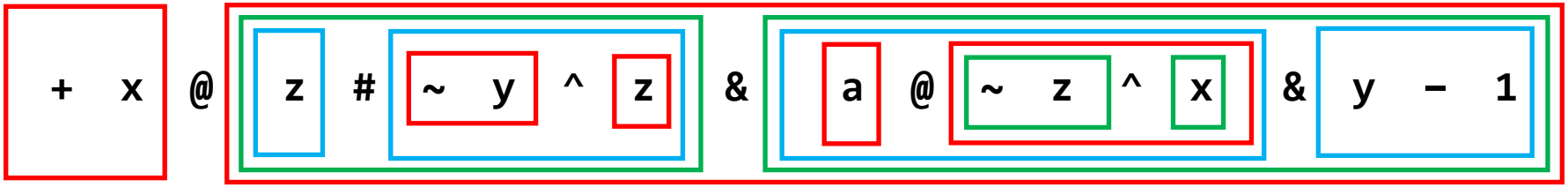



One of the subexpressions in the inner red boxes has more than one operator. Find the operator of that subexpression that's applied last, and its operands:



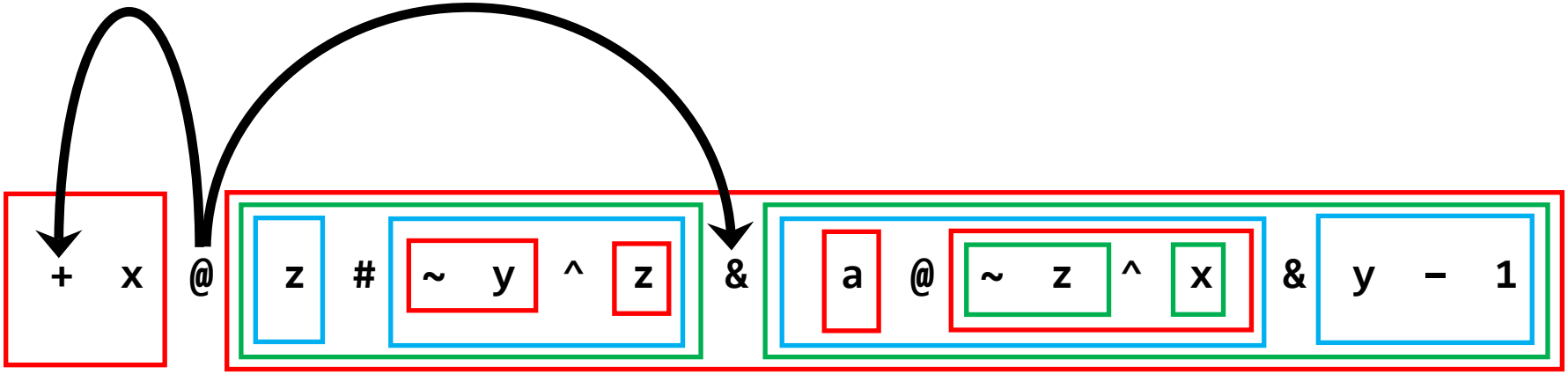
**Example:** Draw the AST of the infix expression

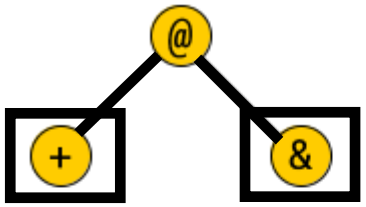
**+ x @ (z # ~ y ^ z) & ( a @ ~ z ^ x) & y – 1**




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:

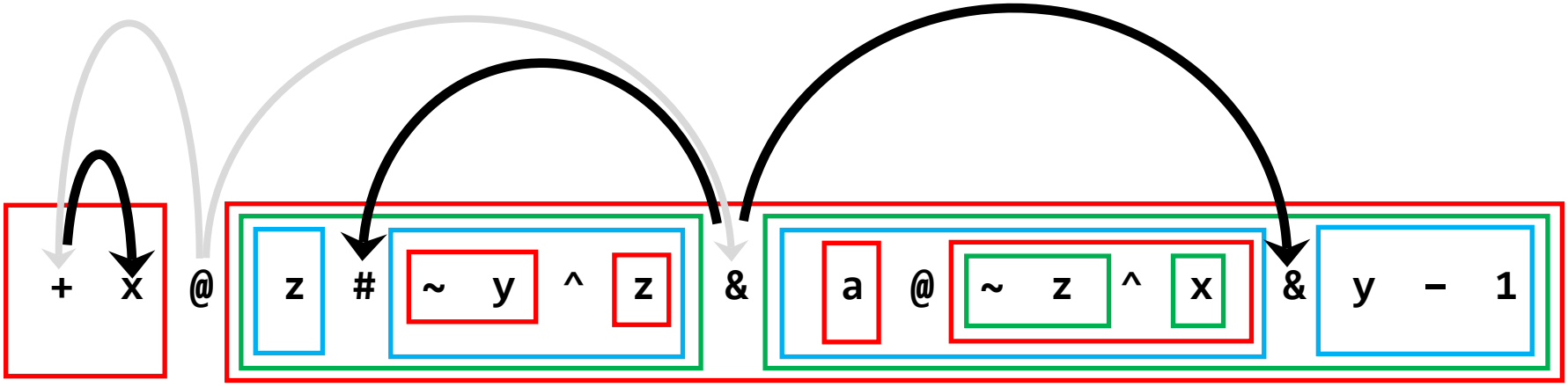
**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



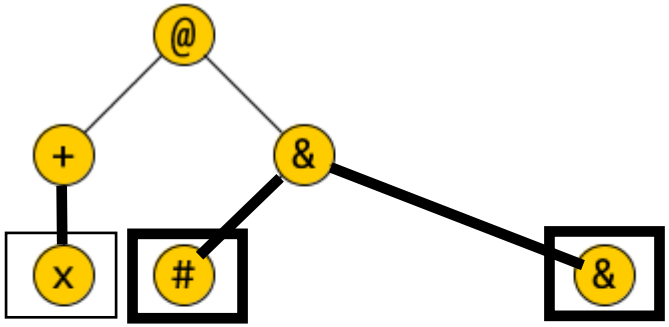




As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:

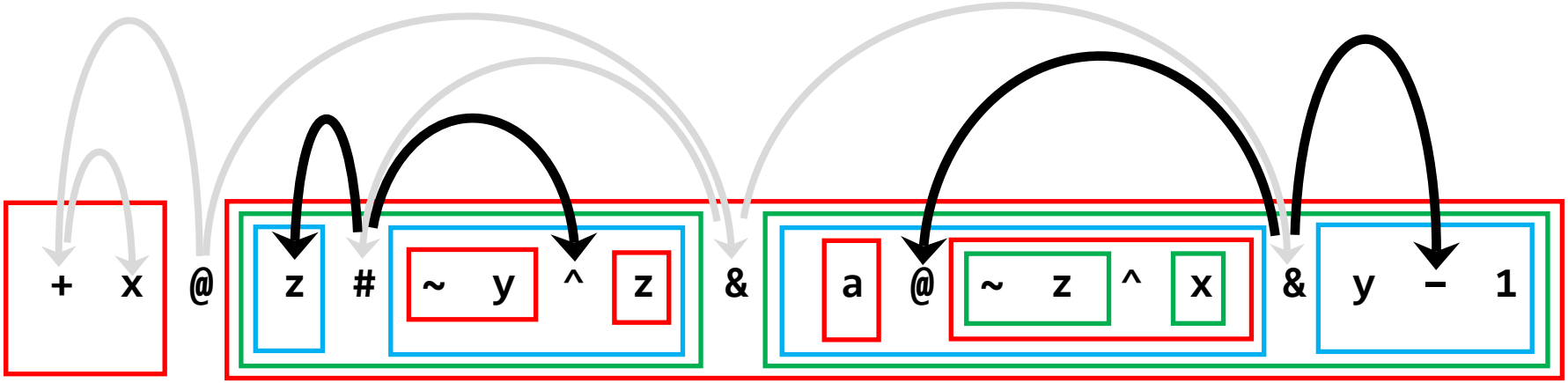
**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



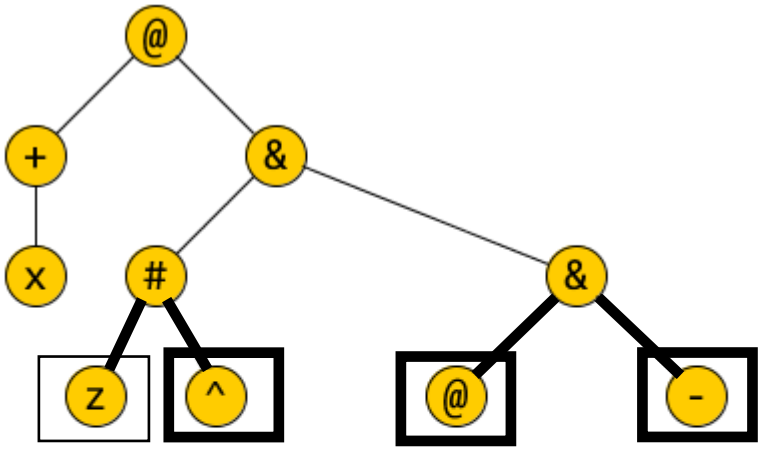

As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:



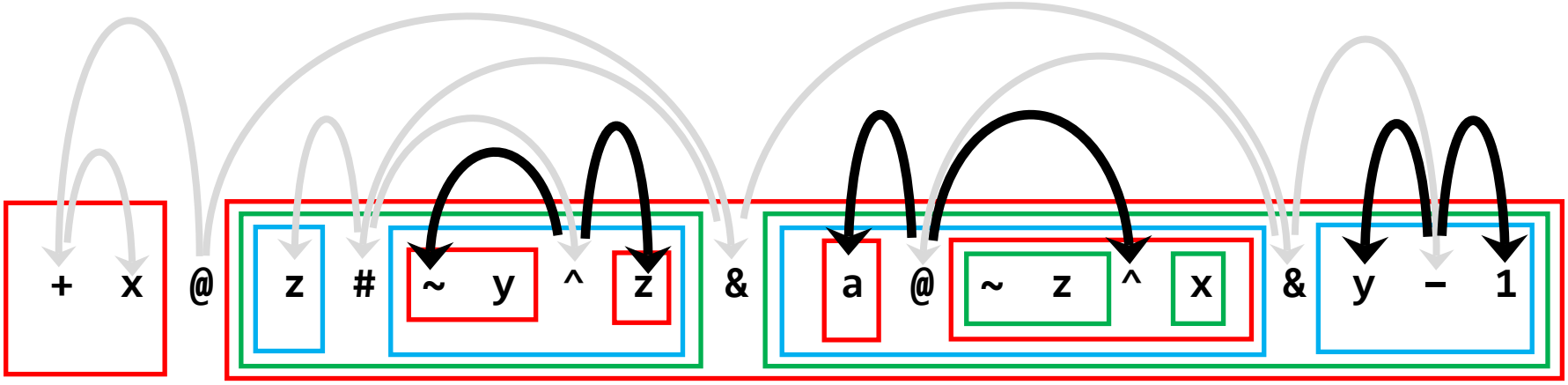

**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



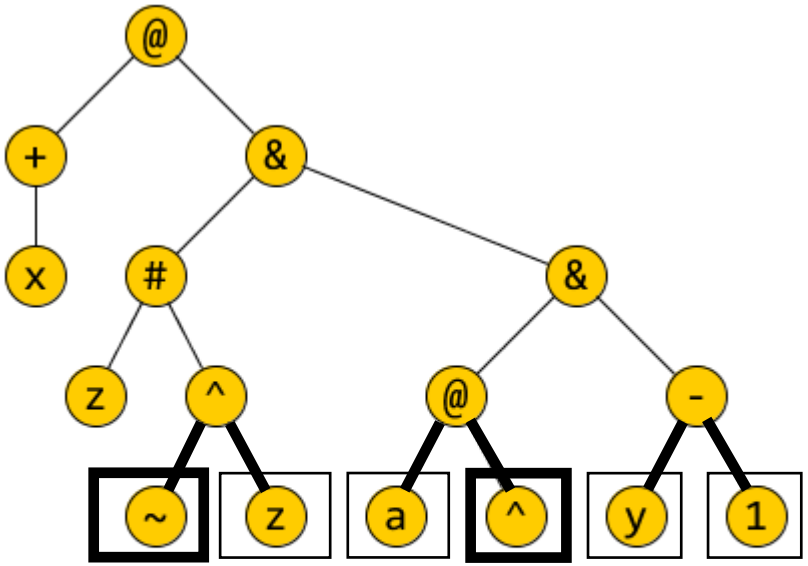

As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:



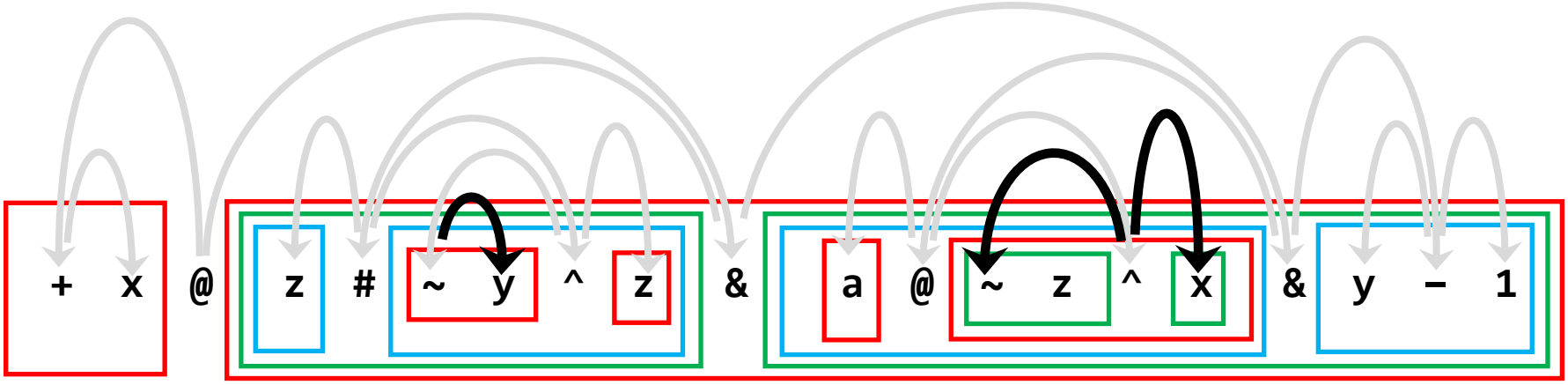

**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



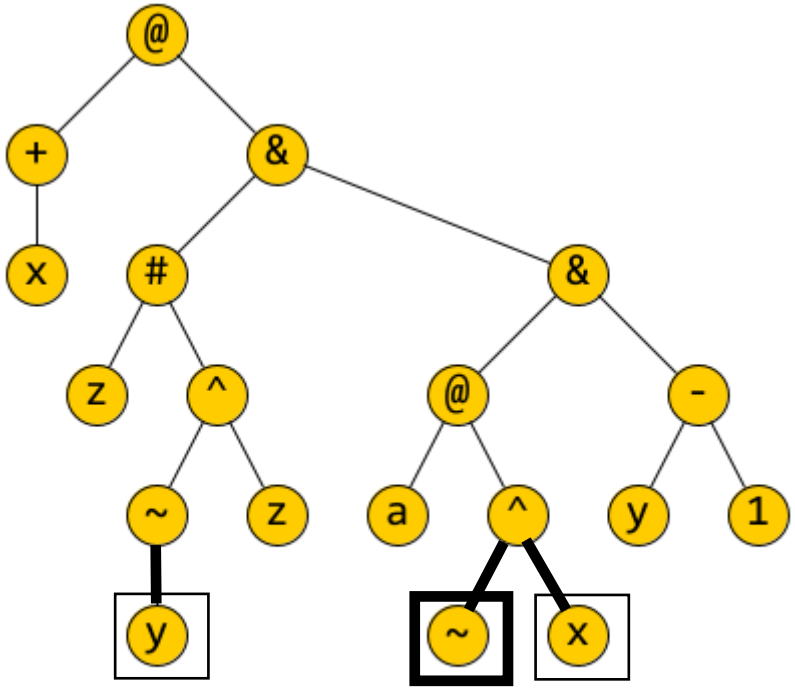

As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:



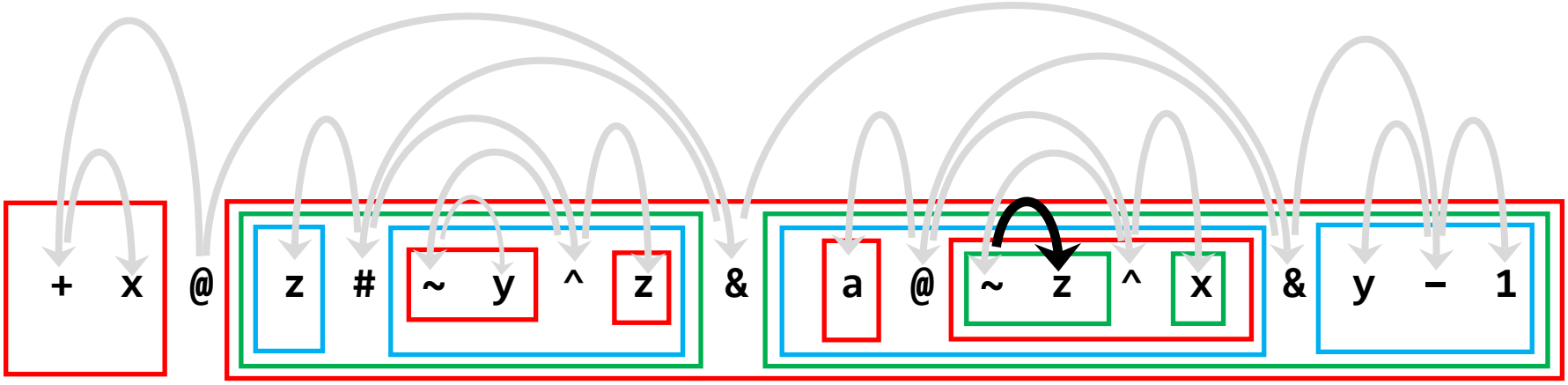

**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



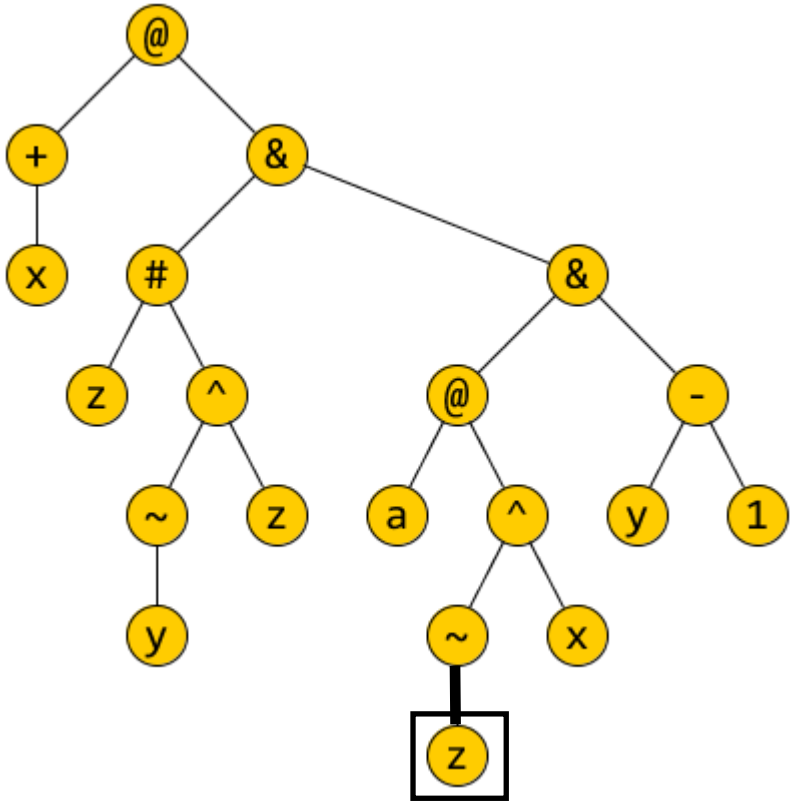

As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:




**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.



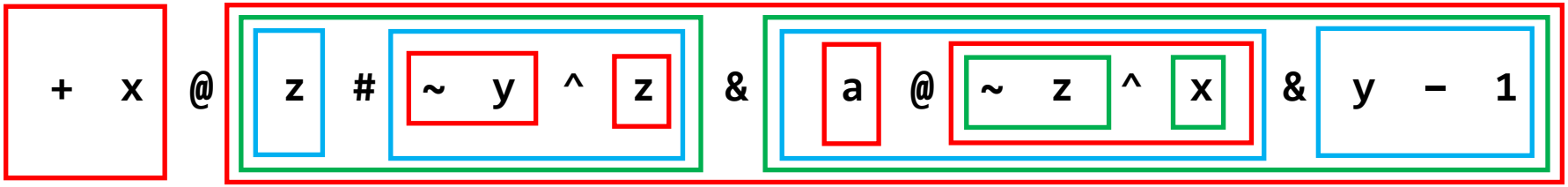

As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:



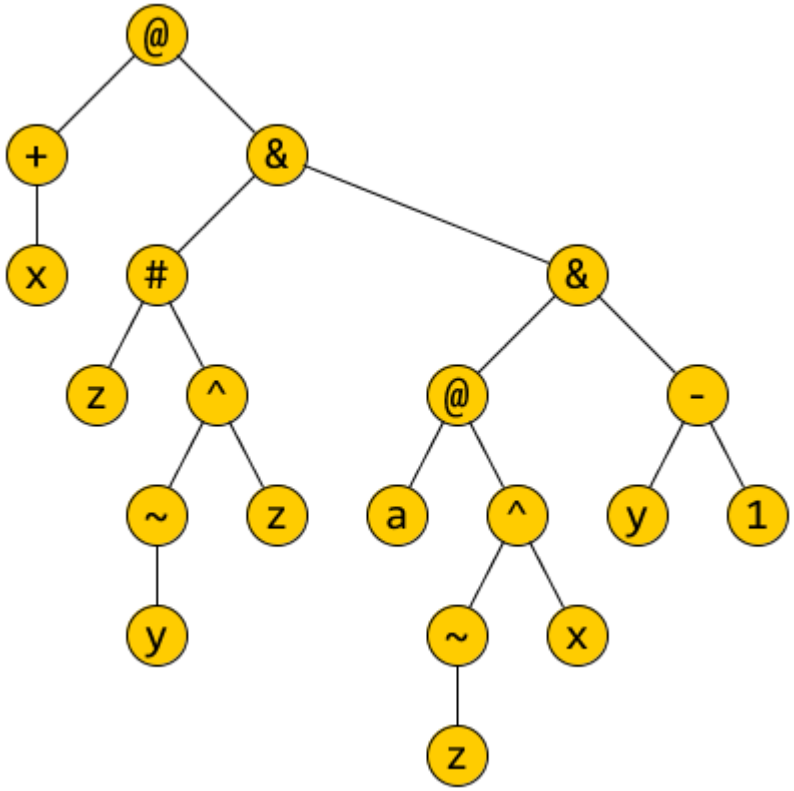

**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.

**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:

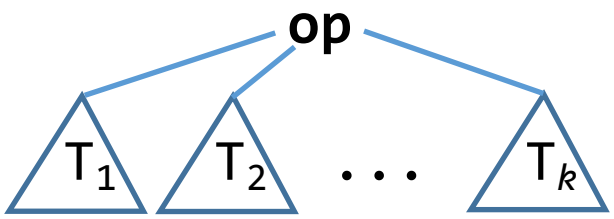


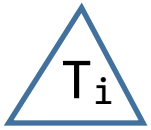

**Recall** that a *k*-ary operator in an AST has ***exactly k children***; constants & variables in an AST are ***leaves***.

**Recall** that 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 $e_i$.

**Example:** Draw the AST of the following expression, which is written in Lisp notation:

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