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.

	prefix unary ops	binary ops	associativity
Class 1			
Class 2			
Class 3			
Class 4			

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

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 $1 \le i < 4$, class i has ***higher*** precedence than class i+1 (so class 1 has highest precedence).

	prefix unary ops	**binary ops**	**associativity**
Class 1			
Class 2			
Class 3			
Class 4			

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

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 $1 \le i < 4$, 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:

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 ≤ 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

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 $1 \le i < 4$, 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

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 $1 \le i < 4$, 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 $1 \le i < 4$, 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.

2.3 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 $1 \le i < 4$, 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.

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 $1 \le i < 4$, 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:

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

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 ≤ 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.

2.3 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 ≤ 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.

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 $1 \le i < 4$, 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.

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:

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.

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 e is an s.v.pre.e.

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 e 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:

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 e 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

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 e 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 e is an s.v.post.e.

Semantics of a Prefix Expression e

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

1.

2.

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 =
2.

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.

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 e is a **prefix** expression, then
   e.value =

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 e 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 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 e is a **prefix** expression, then
   e.value = the result of applying **op** with
   e_i.value as its i^{th} argument ($1 \le i \le k$).

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 e is a **postfix** expression, then
   e.value = the result of applying **op** with
   e_i.value as its i^{th} argument ($1 \le i \le k$).

Some Notable Differences Between Prefix/Postfix and Infix Notations

-
-
-
-

Some Notable Differences Between Prefix/Postfix and Infix Notations

- Prefix and postfix notations are parenthesis-free.
-
-
-

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.

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.

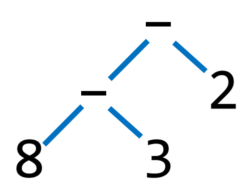
Abstract Syntax Trees

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).

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).

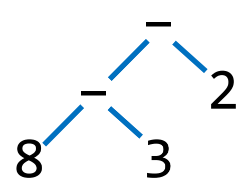
For example, the Lisp expression `(- (- 8 3) 2)` and the two Java expressions `8 - 3 - 2` and `((8 - 3) - 2)` all have the following AST:



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:

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.

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

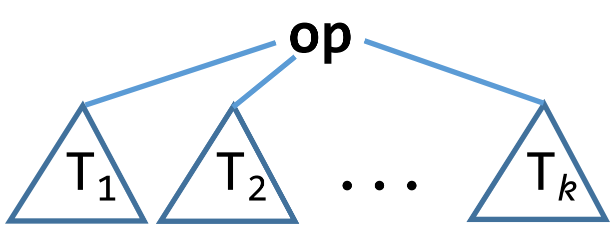
1. If e contains ***no*** operator, then e is equivalent to a variable or constant. In this case e's AST *has just one node*, which is the variable or constant itself.

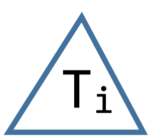
2. In all other cases, let **op** be the operator of e that should be applied *last* when evaluating e, let k be the arity of **op**, and let e_1, ..., e_k be the subexpressions that are the k operands of **op** (where e_i is the ith operand). Then the AST of e is

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 ith operand). Then the AST of e is



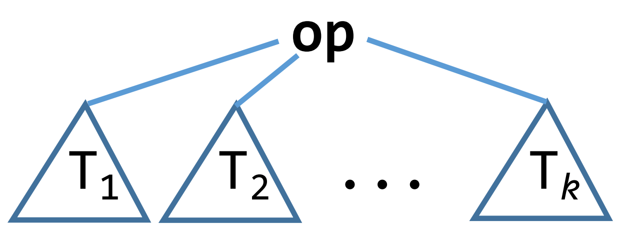
where 

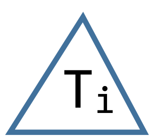
is the AST of the subexpression e_i.

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 <u>*last*</u> 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 ith 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)
```

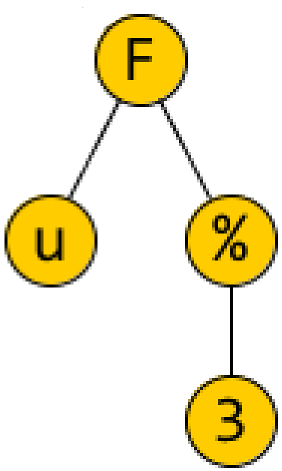
Example: Draw the AST of the following expression, which is written in Lisp notation:

```
(& 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)
```



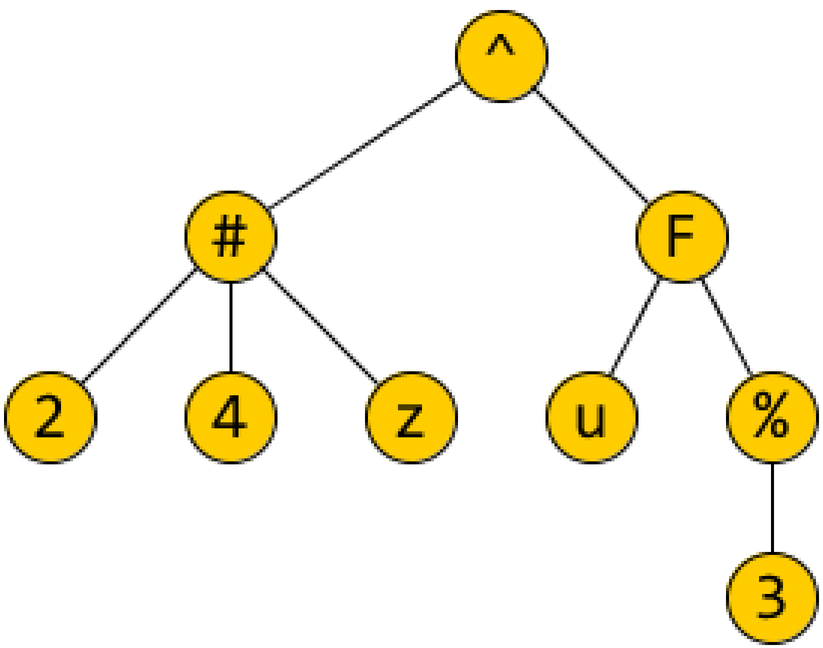
Example: Draw the AST of the following expression, which is written in Lisp notation:

(& 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)



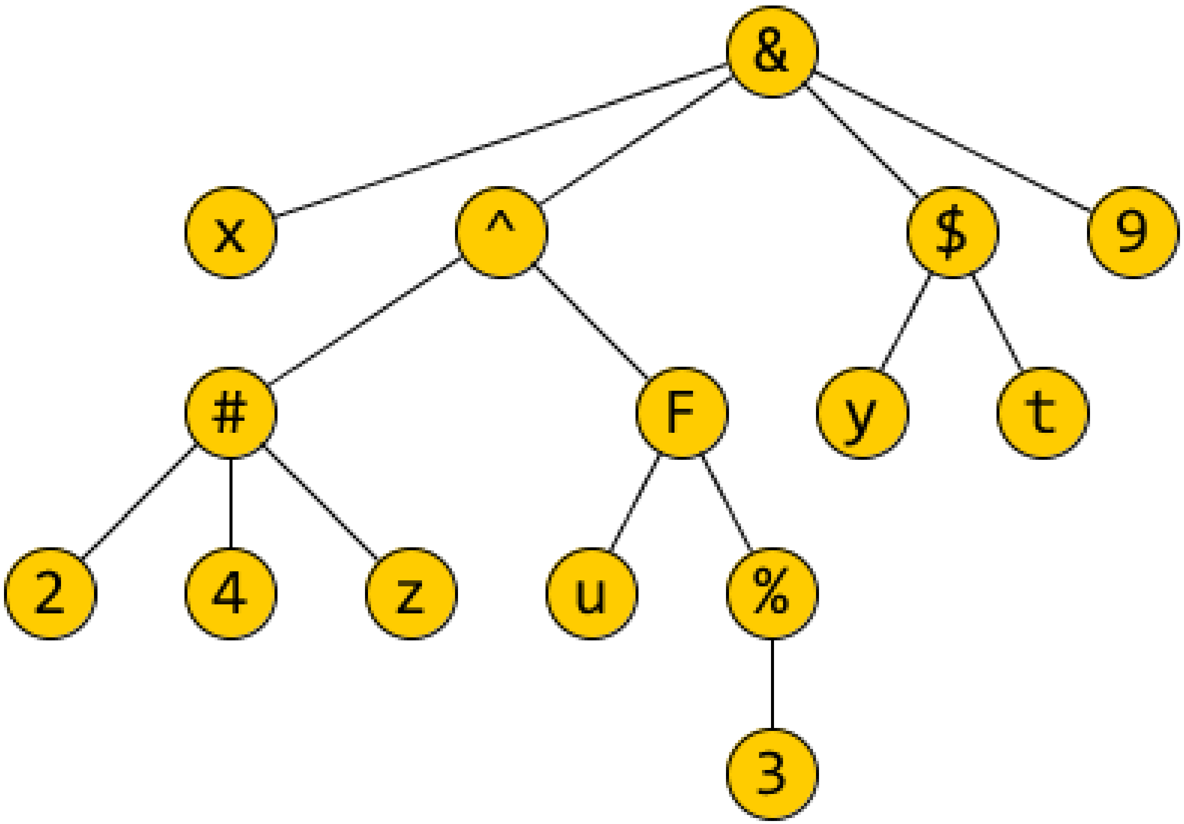
Example: Draw the AST of the following expression, which is written in Lisp notation:

(& 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 &)

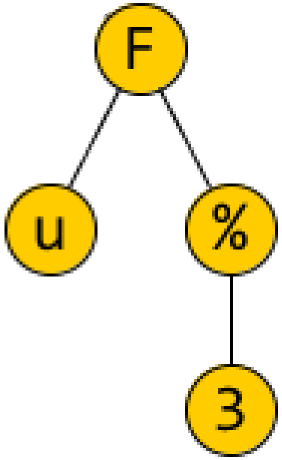
Example: Draw the AST of the following expression, which is written in “rpnLisp” notation:

```
(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 &)
```



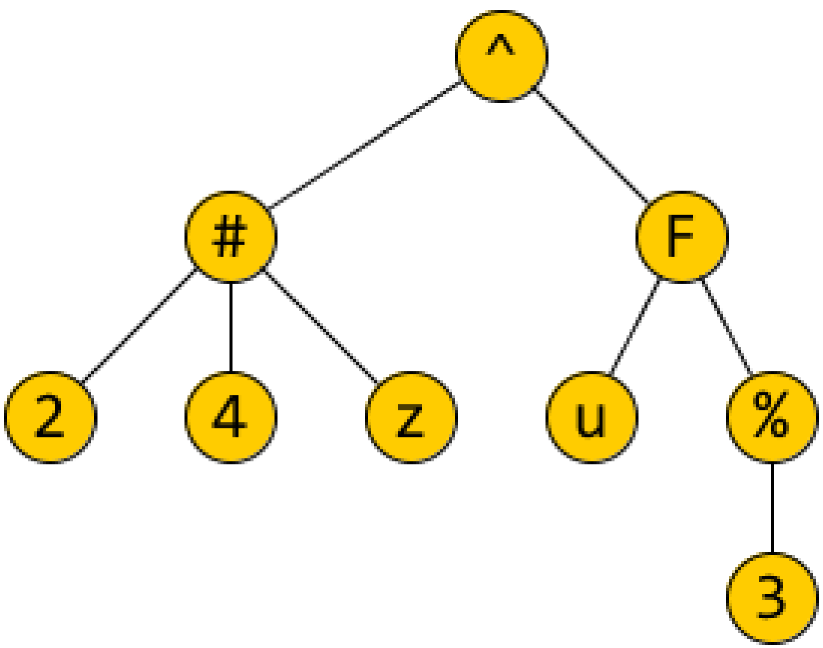
Example: Draw the AST of the following expression, which is written in “rpnLisp” notation:

(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 &)



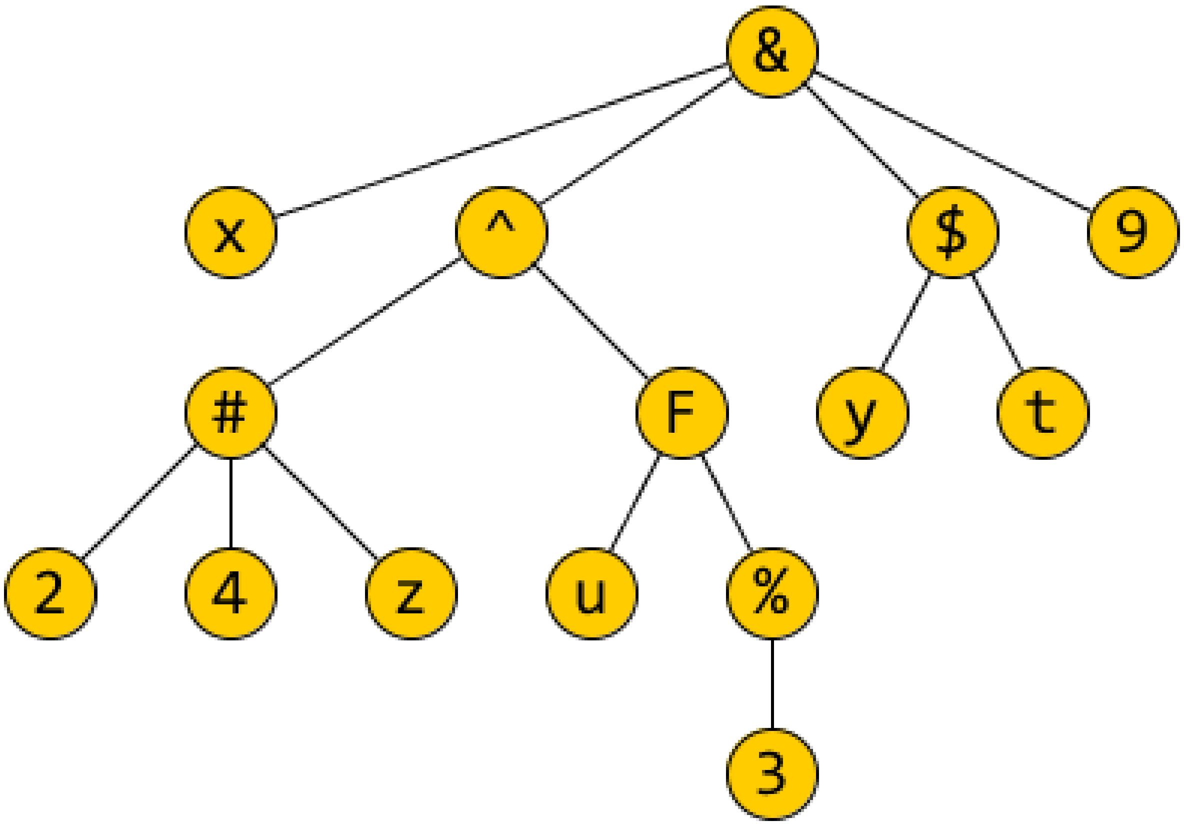
Example: Draw the AST of the following expression, which is written in "rpnLisp" notation:

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

Solution:

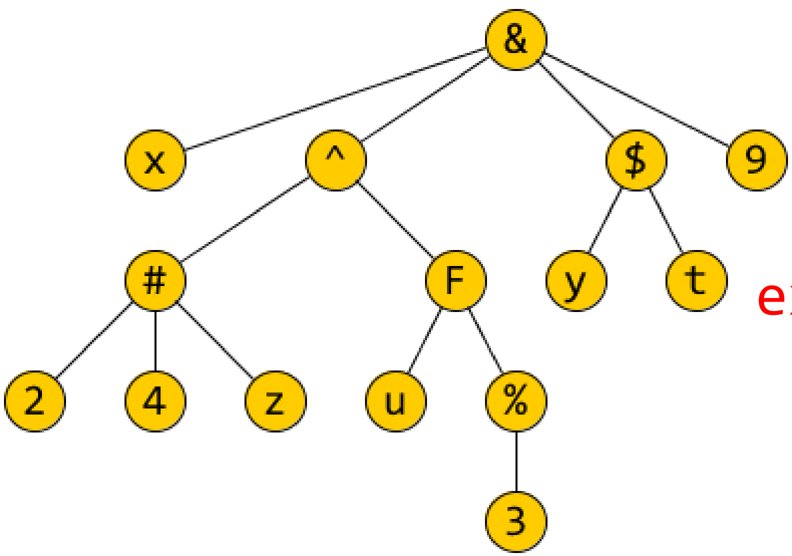
This is the AST of:

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



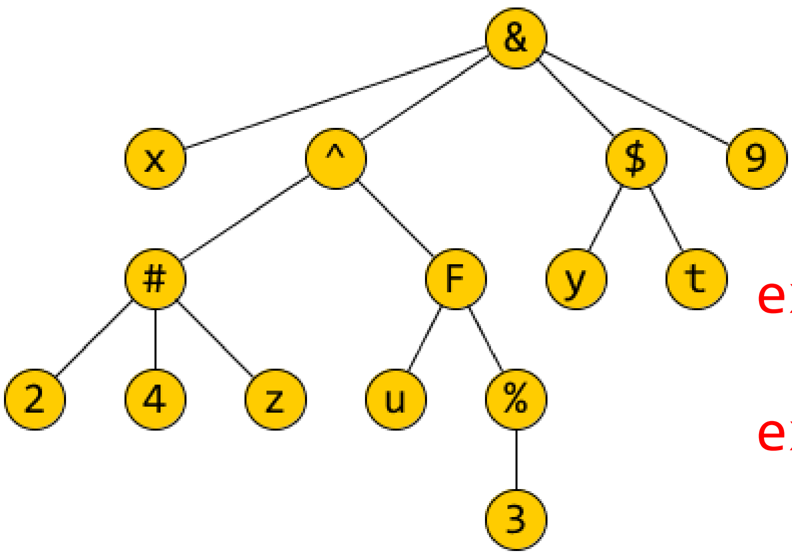
- To draw a prefix expression’s AST, you can write an equivalent Lisp expression and then draw its AST.
- To draw a postfix expression’s AST, you can write an equivalent rpnLisp expression and then draw its AST.

- 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.



expression in ***prefix*** notation:
& x ^ # 2 4 z F u % 3 $ 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.

Solution:

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:

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:

$\boxed{\textbf{+ x}}$ **@** $\boxed{\textbf{(z \# ~ y ^ z) \& (a @ ~ z ^ x) \& y - 1}}$

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.

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

$\boxed{\textbf{+ x}}$ **@** $\boxed{\textbf{(z \# ~ y ^ z) \& (a @ ~ z ^ x) \& y - 1}}$

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

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 $1 \le i < 4$, 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:

$\boxed{\texttt{+ x}}$ **@** $\boxed{\texttt{(z \# \textasciitilde{} y \^{} z) \& (a @ \textasciitilde{} z \^{} x) \& y - 1}}$

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

$\boxed{\texttt{+ x}}$ **@** $\boxed{\boxed{\texttt{(z \# \textasciitilde{} y \^{} z)}} \;\texttt{\&}\; \boxed{\texttt{(a @ \textasciitilde{} z \^{} x) \& y - 1}}}$

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

+ 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.

$\boxed{\textbf{+ x}}$ **@** $\boxed{\boxed{\textbf{(z \# ~ y ^ z)}}\ \textbf{\&}\ \boxed{\textbf{(a @ ~ z ^ x) \& y - 1}}}$

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 ≤ i < 4, class *i* has ***higher*** precedence than class *i*+1.

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

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

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

+ x @ (z # ~ y ^ z) & (a @ ~ z ^ x) & y - 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 ≤ i < 4, class *i* has ***higher*** precedence than class *i*+1.

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

`+ 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.

`+ 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 $1 \le i < 4$, class *i* has ***higher*** precedence than class *i*+1.

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

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.

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

`+ 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

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

Example: Draw the AST of the infix expression

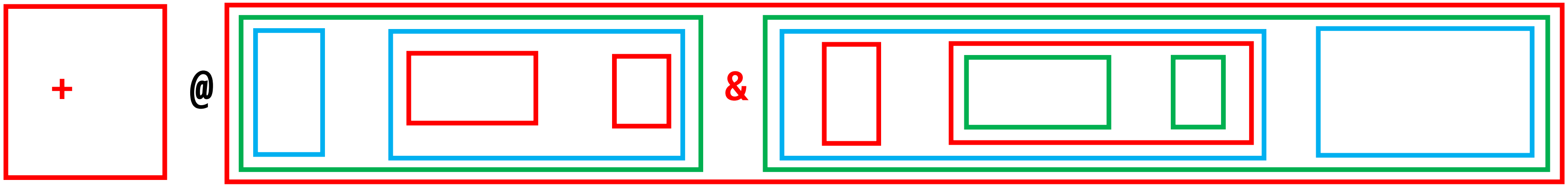
+ x @ (z # ~ y ^ z) & (a @ ~ z ^ x) & y - 1

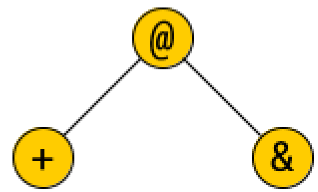
+ x @ z # ~ y ^ z & a @ ~ z ^ x & y - 1

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: Draw the AST of the infix expression

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

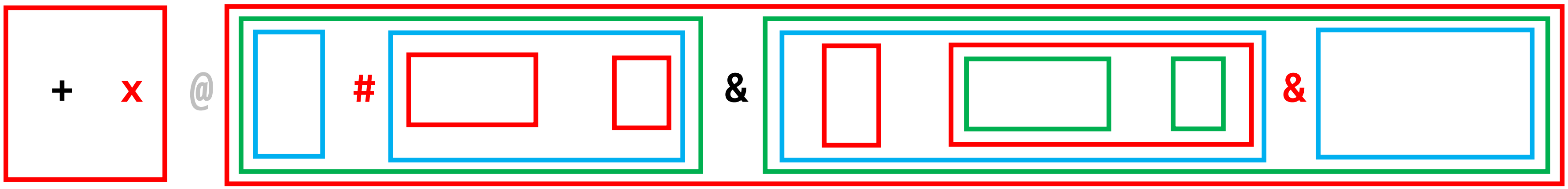


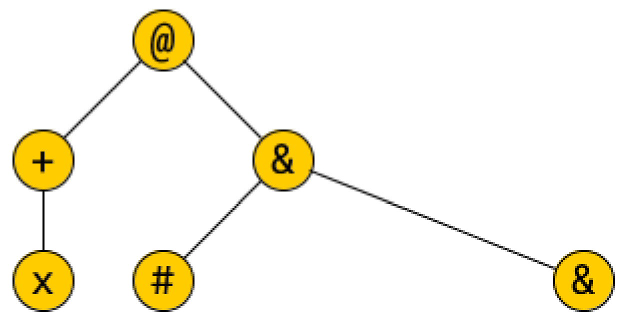


As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:

Example: Draw the AST of the infix expression

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

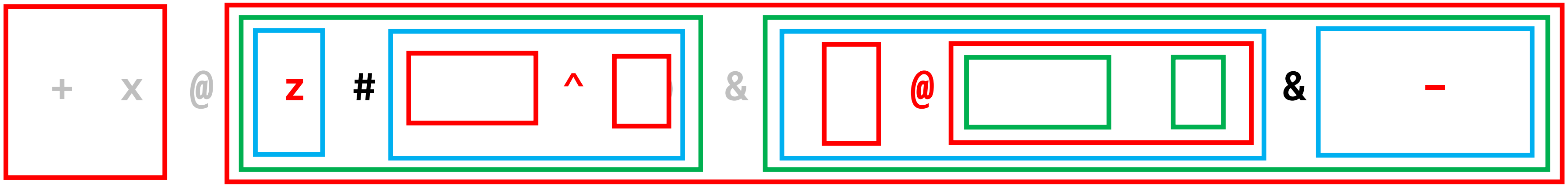




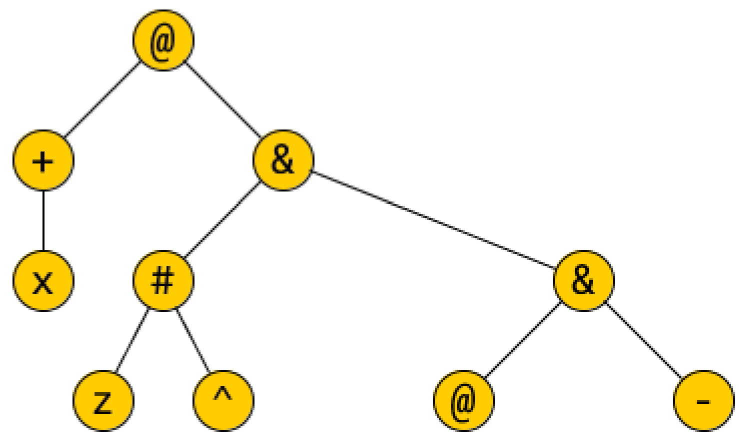
As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:

Example: Draw the AST of the infix expression

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

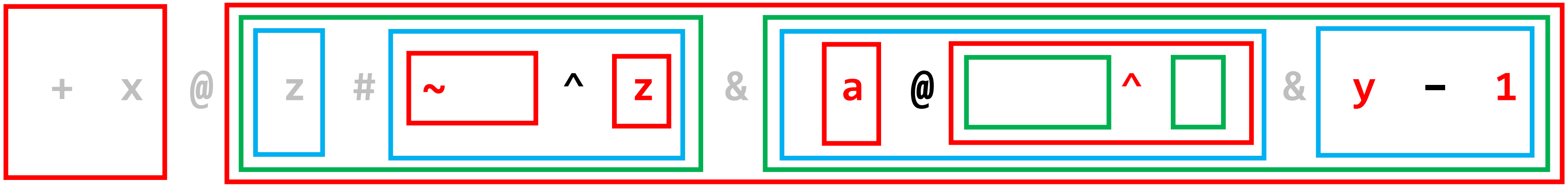


As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:

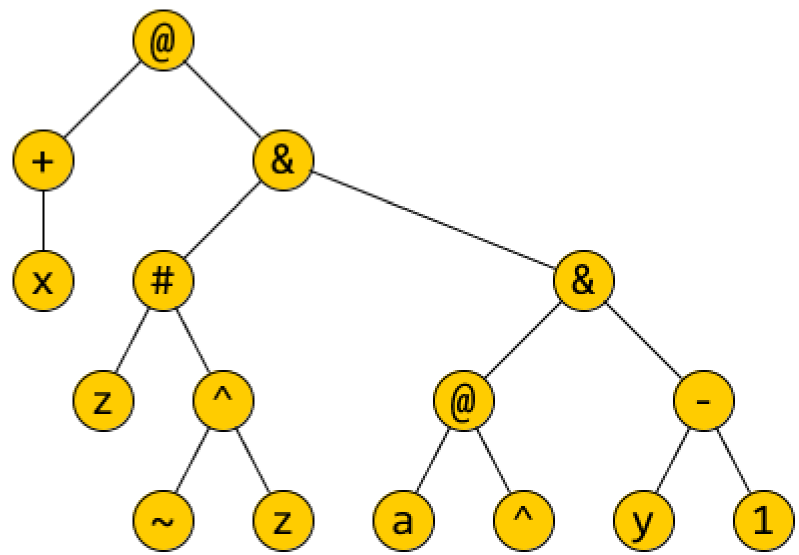


Example: Draw the AST of the infix expression

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

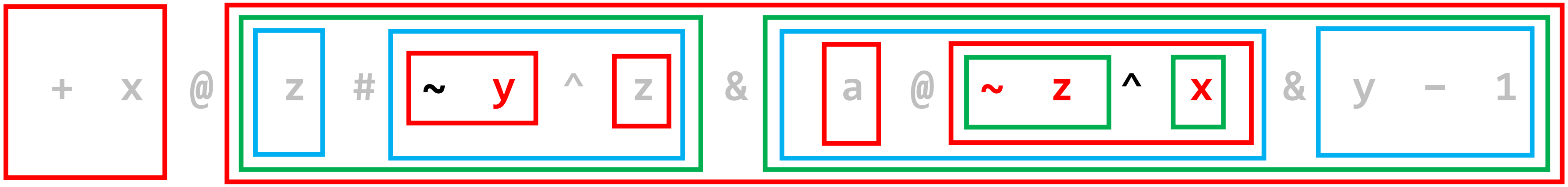


As the subexpressions in the innermost boxes each have at most one operator, it now is straightforward to draw the AST of the entire expression:

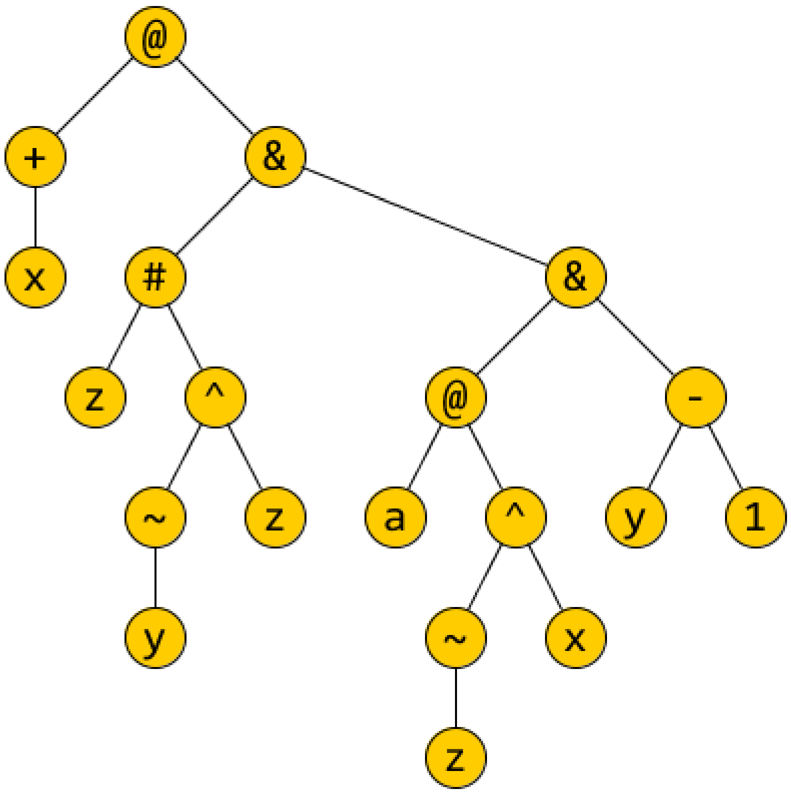


Example: Draw the AST of the infix expression

+ 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:

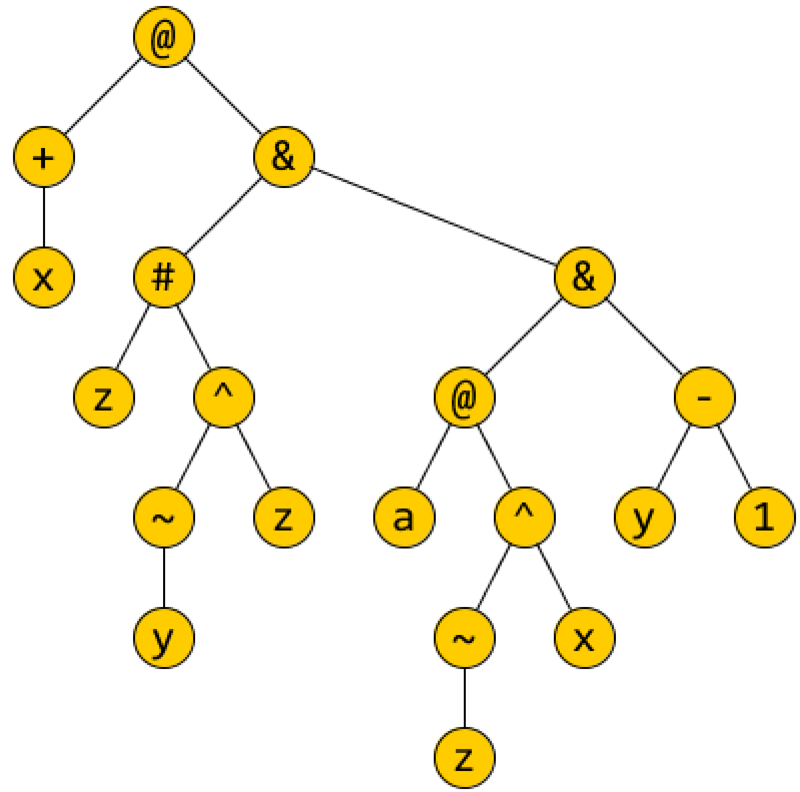


Example: Draw the AST of the infix expression

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

`+ 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:



ASTs of Language Constructs Other Than Expressions

ASTs can also be defined for programming language constructs other than expressions.

They are commonly used to represent the source program during compilation or interpretation.

ASTs of Language Constructs Other Than Expressions

ASTs can also be defined for programming language constructs other than expressions.

They are commonly used to represent the source program during compilation or interpretation.

- The root of an AST for a construct is a node that identifies the kind of construct it is.
-

ASTs of Language Constructs Other Than Expressions

ASTs can also be defined for programming language constructs other than expressions.

They are commonly used to represent the source program during compilation or interpretation.

- The root of an AST for a construct is a node that identifies the kind of construct it is.
- The subtrees of the root are ASTs of substructures whose meanings determine the meaning of the construct.

Example of a Possible AST of a Java Statement

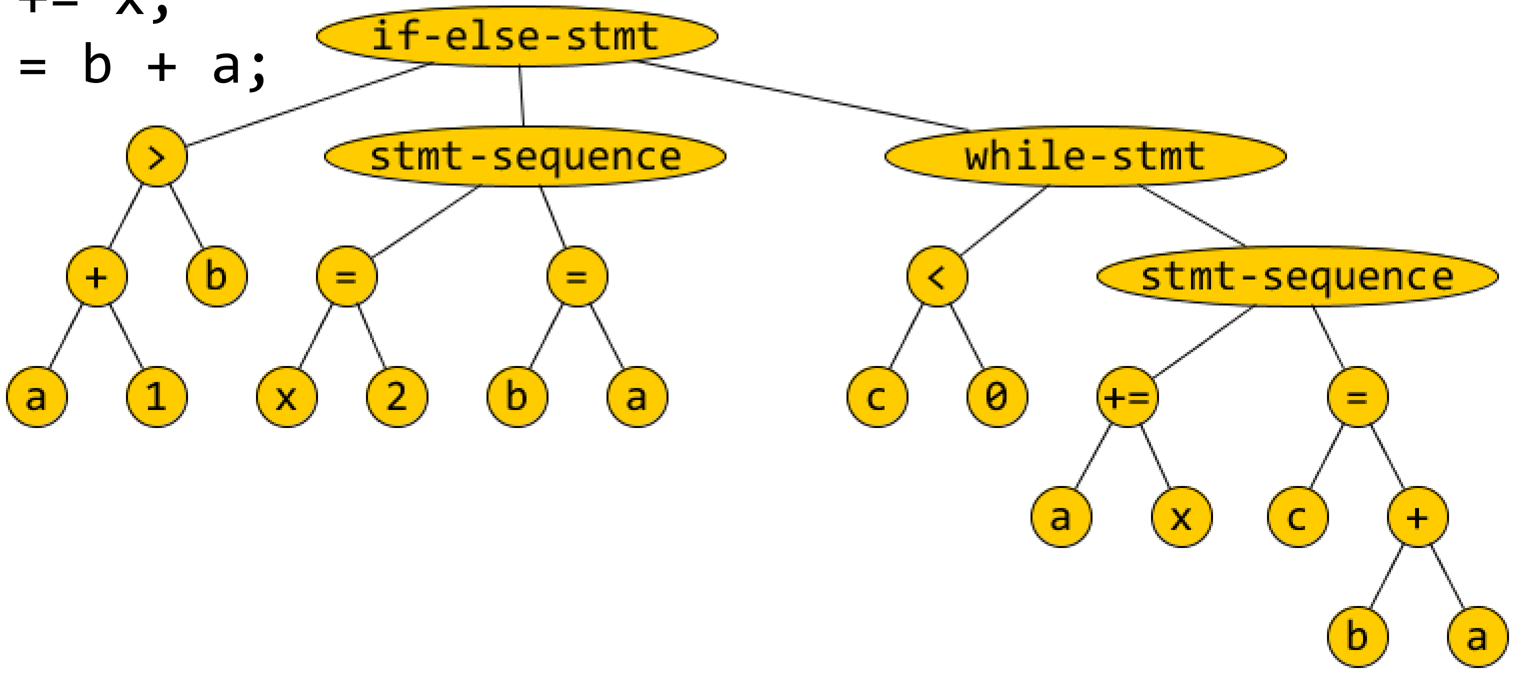
```
if (a+1 > b) {
  x = 2;
  b = a;
}
else {
  while (c < 0) {
    a += x;
    c = b + a;
  }
}
```

can be represented by the following AST:

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:

Evaluation of Postfix Expressions Using a Stack

Postfix expressions can be evaluated as follows:

- Read the expression *from left to right*.

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.

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.

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.

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!

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***.

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = “***rpnLisp***” *notation* *without parentheses*.

Translating Prefix/Postfix Notations to Lisp/"rpnLisp"

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = *"**rpnLisp**" notation* *without parentheses*.

Lisp: (*$_3$ x (-$_2$ (+$_3$ 2 3 y) (*$_2$ w x)) 5)

rpnLisp: (x ((2 3 y +$_3$) (w x *$_2$) -$_2$) 5 *$_3$)

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation without parentheses*.
- ***Postfix*** notation = “***rpnLisp***” *notation without parentheses*.

Lisp: $(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y\ )\ (*_2\ w\ x)\ )\ 5\ )$

Prefix notation: $*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$

rpnLisp: $(\ x\ (\ (2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = *“**rpnLisp**” notation* *without parentheses*.

Lisp:	$(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y\ )\ (*_2\ w\ x)\ )\ 5\ )$
Prefix notation:	$*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$
rpnLisp:	$(\ x\ (\ (2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$
Postfix notation:	$x\ 2\ 3\ y\ +_3\ w\ x\ *_2\ -_2\ 5\ *_3$

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = “***rpnLisp***” *notation* *without parentheses*.

Lisp: $(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y\ )\ (*_2\ w\ x)\ )\ 5\ )$

Prefix notation: $*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$

rpnLisp: $(\ x\ (\ (2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$

Postfix notation: $x\ 2\ 3\ y\ +_3\ w\ x\ *_2\ -_2\ 5\ *_3$

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A.

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = *“**rpnLisp**” notation* *without parentheses*.

Lisp:	$(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y)\ (*_2\ w\ x))\ 5)$
Prefix notation:	$*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$
rpnLisp:	$(x\ ((2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$
Postfix notation:	$x\ 2\ 3\ y\ +_3\ w\ x\ *_2\ -_2\ 5\ *_3$

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Translating Prefix/Postfix Notations to Lisp/"rpnLisp"

Recall:

- ***Prefix*** notation = ***Lisp*** *notation without parentheses*.
- ***Postfix*** notation = *"**rpnLisp**" notation without parentheses*.

Lisp: $(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y)\ (*_2\ w\ x))\ 5)$

Prefix notation: $*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$

rpnLisp: $(x\ ((2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$

Postfix notation: $x\ 2\ 3\ y\ +_3\ w\ x\ *_2\ -_2\ 5\ *_3$

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$ and $\boxed{e_1\ \ldots\ e_k\ \textbf{op}}$ for the Lisp and rpnLisp expressions $(\textbf{op}\ e_1\ \ldots\ e_k)$ and $(e_1\ \ldots\ e_k\ \textbf{op})$.

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write **op** e_1 ... e_k and e_1 ... e_k **op** for the Lisp and rpnLisp expressions (**op** e_1 ... e_k) and (e_1 ... e_k **op**).

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write $\boxed{\mathbf{op}\ e_1\ \ldots\ e_k}$ and $\boxed{e_1\ \ldots\ e_k\ \mathbf{op}}$ for the Lisp and rpnLisp expressions $(\mathbf{op}\ e_1\ \ldots\ e_k)$ and $(e_1\ \ldots\ e_k\ \mathbf{op})$.

The Lisp expression $(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y)\ (*_2\ w\ x))\ 5)$

will be written $\boxed{*_3\ x\ \boxed{-_2\ \boxed{+_3\ 2\ 3\ y}\ \boxed{*_2\ w\ x}}\ 5}$

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write $\boxed{\mathbf{op}\ e_1\ \ldots\ e_k}$ and $\boxed{e_1\ \ldots\ e_k\ \mathbf{op}}$ for the Lisp and rpnLisp expressions ($\mathbf{op}\ e_1\ \ldots\ e_k$) and ($e_1\ \ldots\ e_k\ \mathbf{op}$).

The Lisp expression $(*_3\ \ x\ (-_2\ (+_3\ 2\ 3\ y)\ (*_2\ w\ x))\ 5)$

will be written $\boxed{*_3\ \ x\ \ \boxed{-_2\ \ \boxed{+_3\ 2\ 3\ y}\ \ \boxed{*_2\ w\ x}}\ \ 5}$

The rpnLisp expression $(x\ ((2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$

will be written $\boxed{x\ \ \boxed{\boxed{2\ 3\ y\ +_3}\ \ \boxed{w\ x\ *_2}\ \ -_2}\ \ 5\ *_3}$

We can use a stack as follows to translate a postfix expression to “rpnLisp”:

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

We can use a stack as follows to translate a postfix expression to “rpnLisp”:

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the “rpnLisp” equivalent of the postfix expression will be the only thing on the stack.

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
    - *Pop* off *k* expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

STACK:

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

STACK: **x**

We can use a stack as follows to translate a postfix expression to “rpnLisp”:

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k \textbf{ op}}$.

After the entire expression has been processed in this way, the “rpnLisp” equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: **x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** **2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

STACK: **x 2**

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \ \ 2 \ \ 3 \ \ +_3 \ \ y \ \ -_2 \ \ u \ \ x \ \ 5 \ \ *_2 \ \ *_3 \ \ -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{3 \ \ +_3 \ \ y \ \ -_2 \ \ u \ \ x \ \ 5 \ \ *_2 \ \ *_3 \ \ -_1}$

STACK: **x 2 3**

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $x \ \ 2 \ \ 3 \ \ +_3 \ \ y \ \ -_2 \ \ u \ \ x \ \ 5 \ \ *_2 \ \ *_3 \ \ -_1$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $+_3 \ \ y \ \ -_2 \ \ u \ \ x \ \ 5 \ \ *_2 \ \ *_3 \ \ -_1$

STACK: $\boxed{x \ \ 2 \ \ 3 \ \ +_3}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \;\; 2 \;\; 3 \;\; +_3 \;\; y \;\; -_2 \;\; u \;\; x \;\; 5 \;\; *_2 \;\; *_3 \;\; -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{y \;\; -_2 \;\; u \;\; x \;\; 5 \;\; *_2 \;\; *_3 \;\; -_1}$

STACK: $\boxed{\mathbf{x \;\; 2 \;\; 3 \;\; +_3}}$ $\mathbf{y}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k\ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x\quad 2\quad 3\quad +_3\quad y\quad -_2\quad u\quad x\quad 5\quad *_2\quad *_3\quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{-_2\quad u\quad x\quad 5\quad *_2\quad *_3\quad -_1}$

STACK: $\boxed{\boxed{\mathbf{x\quad 2\quad 3\quad +_3}}\quad \mathbf{y\quad -_2}}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \ 2 \ 3 \ +_3 \ y \ -_2 \ u \ x \ 5 \ *_2 \ *_3 \ -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{u \ x \ 5 \ *_2 \ *_3 \ -_1}$

STACK: $\boxed{\boxed{\mathbf{x \ 2 \ 3 \ +_3}} \ \mathbf{y \ -_2}} \ \mathbf{u}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \textbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: **x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** x **5 $*_2$ $*_3$ $-_1$**

STACK: $\boxed{\boxed{\mathbf{x\ \ 2\ \ 3\ \ +_3}}\ \mathbf{y\ \ -_2}}$ **u x**

We can use a stack as follows to translate a postfix expression to “rpnLisp”:

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the “rpnLisp” equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $\mathbf{5 \quad *_2 \quad *_3 \quad -_1}$

STACK: $\boxed{\boxed{\mathbf{x \quad 2 \quad 3 \quad +_3}} \quad \mathbf{y \quad -_2}} \quad \mathbf{u \quad x \quad 5}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $*_2 \quad *_3 \quad -_1$

STACK: $\boxed{\boxed{x \quad 2 \quad 3 \quad +_3} \quad y \quad -_2} \quad u \quad \boxed{x \quad 5 \quad *_2}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $*_3 \quad -_1$

STACK: $\boxed{\boxed{\boxed{x \quad 2 \quad 3 \quad +_3} \quad y \quad -_2} \quad u \quad \boxed{x \quad 5 \quad *_2} \quad *_3}$

We can use a stack as follows to translate a postfix expression to “rpnLisp”:

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_k, ..., e_1.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the “rpnLisp” equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** $-_1$

STACK: $\boxed{\boxed{\boxed{\boxed{x \quad 2 \quad 3 \quad +_3} \quad y \quad -_2} \quad u \quad \boxed{x \quad 5 \quad *_2} \quad *_3} \quad -_1}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions $e_k, \ldots, e_1$.
    - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

rpnLisp equivalent of the expression

$$\boxed{\boxed{\boxed{\boxed{x \quad 2 \quad 3 \quad +_3} \quad y \quad -_2} \quad u \quad \boxed{x \quad 5 \quad *_2} \quad *_3} \quad -_1}$$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
    - *Pop* off *k* expressions e_1, ..., e_k.
    - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\mathbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$

STACK:

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y\quad *_2\quad w\quad x\quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y\quad *_2\quad w\quad x\quad 5$

STACK: **5**

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$*_3$ x $-_2$ $+_3$ 2 3 y $*_2$ w x 5

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3$ x $-_2$ $+_3$ 2 3 y $*_2$ w x

STACK: x 5

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w$

STACK: **w x 5**

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \dots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2$

STACK: $\boxed{*_2\ w\ x}\quad 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y\quad *_2\quad w\quad x\quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y$

STACK: $y\quad \boxed{*_2\quad w\quad x}\quad 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions e_1, ..., e_k.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$*_3$ x $-_2$ $+_3$ 2 3 y $*_2$ w x 5

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3$ x $-_2$ $+_3$ 2 3

STACK: 3 y $\boxed{*_2\ \mathtt{w}\ \mathtt{x}}$ 5

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\textbf{op } e_1 \ldots e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2$

STACK: $2 \quad 3 \quad y \quad \boxed{*_2 \quad w \quad x} \quad 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions e_1, ..., e_k.
    - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3$

STACK: $\boxed{+_3\ 2\ 3\ y}\ \boxed{*_2\ w\ x}\ 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y\quad *_2\quad w\quad x\quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3\quad x\quad -_2$

STACK: $\boxed{-_2\ \boxed{+_3\ 2\ 3\ y}\ \boxed{*_2\ w\ x}}\quad 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions e_1, ..., e_k.
    - *Push* the Lisp expression $\boxed{\mathbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3\quad x\quad -_2\quad +_3\quad 2\quad 3\quad y\quad *_2\quad w\quad x\quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3$ x

STACK: $x\quad \boxed{-_2\ \boxed{+_3\ 2\ 3\ y}\ \boxed{*_2\ w\ x}}\quad 5$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3$

STACK: $\boxed{*_3 \quad x \quad \boxed{-_2 \quad \boxed{+_3 \quad 2 \quad 3 \quad y} \quad \boxed{*_2 \quad w \quad x}} \quad 5}$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a *k*-ary operator **op** is seen:
  - *Pop* off *k* expressions $e_1, \ldots, e_k$.
  - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

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

Example Translate the following prefix expression into Lisp.

$$*_3\ \ x\ \ -_2\ \ +_3\ \ 2\ \ 3\ \ y\ \ *_2\ \ w\ \ x\ \ 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:**

STACK: $\boxed{*_3\ \ x\ \ \boxed{-_2\ \ \boxed{+_3\ \ 2\ \ 3\ \ y}\ \ \boxed{*_2\ \ w\ \ x}}\ \ 5}$

Lisp equivalent of the expression

Note that:

- The structure of the Lisp / rpnLisp equivalent of a prefix / postfix expression does ***not*** depend on the names and semantics of the operators, but only depends on the *arities* of the operators.

Note that:

- The structure of the Lisp / rpnLisp equivalent of a prefix / postfix expression does ***not*** depend on the names and semantics of the operators, but only depends on the *arities* of the operators.

For example, the problem

Translate the following postfix expression into rpnLisp:

x 2 3 $@_3$ y $\#_2$ u x 5 $\hat{}_2$ $!_3$ $\sim_1$

Here $@_3$ and $!_3$ are 3-ary, $\hat{}_2$ and $\#_2$ are binary, and $\sim_1$ is unary.

is essentially equivalent to the problem

Note that:

- The structure of the Lisp / rpnLisp equivalent of a prefix / postfix expression does ***not*** depend on the names and semantics of the operators, but only depends on the *arities* of the operators.

For example, the problem

Translate the following postfix expression into rpnLisp:

x 2 3 $@_3$ y $\#_2$ u x 5 $\hat{}_2$ $!_3$ $\sim_1$

Here $@_3$ and $!_3$ are 3-ary, $\hat{}_2$ and $\#_2$ are binary, and $\sim_1$ is unary.

is essentially equivalent to the problem

Translate the following postfix expression into rpnLisp:

x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

that we solved above:

Note that:

- The structure of the Lisp / rpnLisp equivalent of a prefix / postfix expression does ***not*** depend on the names and semantics of the operators, but only depends on the *arities* of the operators.

For example, the problem

> Translate the following postfix expression into rpnLisp:
>
> **x 2 3 $@_3$ y $\#_2$ u x 5 $\wedge_2$ $!_3$ $\sim_1$**
>
> Here $@_3$ and $!_3$ are 3-ary, $\wedge_2$ and $\#_2$ are binary, and $\sim_1$ is unary.

is essentially equivalent to the problem

> Translate the following postfix expression into rpnLisp:
>
> **x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**
>
> Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

that we solved above: Substituting $@_3$, $!_3$, $\wedge_2$, $\#_2$, and $\sim_1$ for $+_3$, $*_3$, $*_2$, $-_2$, and $-_1$ in our solution to the latter problem gives a solution to the former problem.

Functions That Return Functions as Their Results

7.16 FUNCTIONS THAT MAKE FUNCTIONS From Touretzky's book

It is possible to write a function whose value is another function. Suppose we want to make a function that returns true if its input is greater than a certain number N. We can make this function by constructing a lambda expression that refers to N, and returning that lambda expression:

```
(defun make-greater-than-predicate (n)
  #'(lambda (x) (> x n)))
```

Functions That Return Functions as Their Results

7.16 FUNCTIONS THAT MAKE FUNCTIONS

From Touretzky’s book

It is possible to write a function whose value is another function. Suppose we want to make a function that returns true if its input is greater than a certain number N. We can make this function by constructing a lambda expression that refers to N, and returning that lambda expression:

```
(defun make-greater-than-predicate (n)
  #'(lambda (x) (> x n)))
```

The value returned by MAKE-GREATER-THAN-PREDICATE will be a lexical closure. We can store this value away somewhere, or pass it as an argument to FUNCALL or any applicative operator.

```
> (setf pred (make-greater-than-predicate 3))
#<Lexical-closure 7315225>
```

Functions That Return Functions as Their Results

7.16 FUNCTIONS THAT MAKE FUNCTIONS

From Touretzky's book

It is possible to write a function whose value is another function. Suppose we want to make a function that returns true if its input is greater than a certain number N. We can make this function by constructing a lambda expression that refers to N, and returning that lambda expression:

```
(defun make-greater-than-predicate (n)
  #'(lambda (x) (> x n)))
```

The value returned by MAKE-GREATER-THAN-PREDICATE will be a lexical closure. We can store this value away somewhere, or pass it as an argument to FUNCALL or any applicative operator.

```
> (setf pred (make-greater-than-predicate 3))
#<Lexical-closure 7315225>

(funcall pred 2)  ⇒  nil

(funcall pred 5)  ⇒  t
```

Functions That Return Functions as Their Results

7.16 FUNCTIONS THAT MAKE FUNCTIONS

From Touretzky's book

It is possible to write a function whose value is another function. Suppose we want to make a function that returns true if its input is greater than a certain number N. We can make this function by constructing a lambda expression that refers to N, and returning that lambda expression:

```
(defun make-greater-than-predicate (n)
  #'(lambda (x) (> x n)))
```

The value returned by MAKE-GREATER-THAN-PREDICATE will be a lexical closure. We can store this value away somewhere, or pass it as an argument to FUNCALL or any applicative operator.

```
> (setf pred (make-greater-than-predicate 3))
#<Lexical-closure 7315225>

(funcall pred 2)  ⇒  nil

(funcall pred 5)  ⇒  t
```

Any function that takes a function as an argument or returns a function as its result is called a ***higher-order function***.

More Examples of Functions That Return Functions

A function that takes two single argument functions as its arguments and returns their composition:

```
[1]> (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
COMPOSE
[2]> (funcall (compose #'sqrt #'car) '(4 dog cat mouse))
2
```

More Examples of Functions That Return Functions

A function that takes two single argument functions as its arguments and returns their composition:

```
[1]> (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
COMPOSE
[2]> (funcall (compose #'sqrt #'car) '(4 dog cat mouse))
2
[3]> (mapcar (compose #'sqrt #'car) '((4 dog) (9 cat) (0 mouse)))
(2 3 0)
```

More Examples of Functions That Return Functions

A function that takes two single argument functions as its arguments and returns their composition:

```
[1]> (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
COMPOSE
[2]> (funcall (compose #'sqrt #'car) '(4 dog cat mouse))
2
[3]> (mapcar (compose #'sqrt #'car) '((4 dog) (9 cat) (0 mouse)))
(2 3 0)
[4]> (funcall (compose #'cdr #'car) '((4 dog) (9 cat) (0 mouse)))
(DOG)
[5]> (funcall (compose #'car #'cdr) '((4 dog) (9 cat) (0 mouse)))
(9 CAT)
```

More Examples of Functions That Return Functions

A function that takes two single argument functions as its arguments and returns their composition:

```
[1]> (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
COMPOSE
[2]> (funcall (compose #'sqrt #'car) '(4 dog cat mouse))
2
[3]> (mapcar (compose #'sqrt #'car) '((4 dog) (9 cat) (0 mouse)))
(2 3 0)
[4]> (funcall (compose #'cdr #'car) '((4 dog) (9 cat) (0 mouse)))
(DOG)
[5]> (funcall (compose #'car #'cdr) '((4 dog) (9 cat) (0 mouse)))
(9 CAT)
```

A function that takes a 2-argument function h as argument and returns a function h′ such that (h′ x y) = (h y x):

```
[6]> (defun flip-arguments (h) (lambda (x y) (funcall h y x)))
FLIP-ARGUMENTS
[7]> (funcall (flip-arguments #'append) '(A B C) '(D E F G))
(D E F G A B C)
```

More Examples of Functions That Return Functions

A function that takes two single argument functions as its arguments and returns their composition:

```
[1]> (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
COMPOSE
[2]> (funcall (compose #'sqrt #'car) '(4 dog cat mouse))
2
[3]> (mapcar (compose #'sqrt #'car) '((4 dog) (9 cat) (0 mouse)))
(2 3 0)
[4]> (funcall (compose #'cdr #'car) '((4 dog) (9 cat) (0 mouse)))
(DOG)
[5]> (funcall (compose #'car #'cdr) '((4 dog) (9 cat) (0 mouse)))
(9 CAT)
```

A function that takes a 2-argument function h as argument and returns a function h′ such that (h′ x y) = (h y x):

```
[6]> (defun flip-arguments (h) (lambda (x y) (funcall h y x)))
FLIP-ARGUMENTS
[7]> (funcall (flip-arguments #'append) '(A B C) '(D E F G))
(D E F G A B C)
[8]> (funcall (flip-arguments #'cons) '(A B C) '(D E F G))
((D E F G) A B C)
[9]> (funcall (flip-arguments #'list) '(A B C) '(D E F G))
((D E F G) (A B C))
[10]>
```