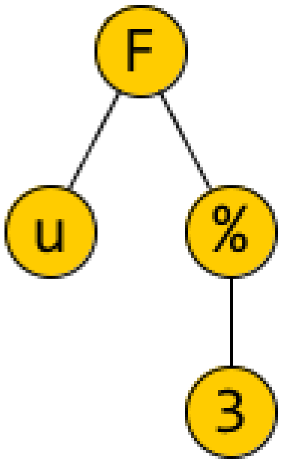
Example: Draw the AST of the following expression, which is written in Lisp notation:

`(& 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)`



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

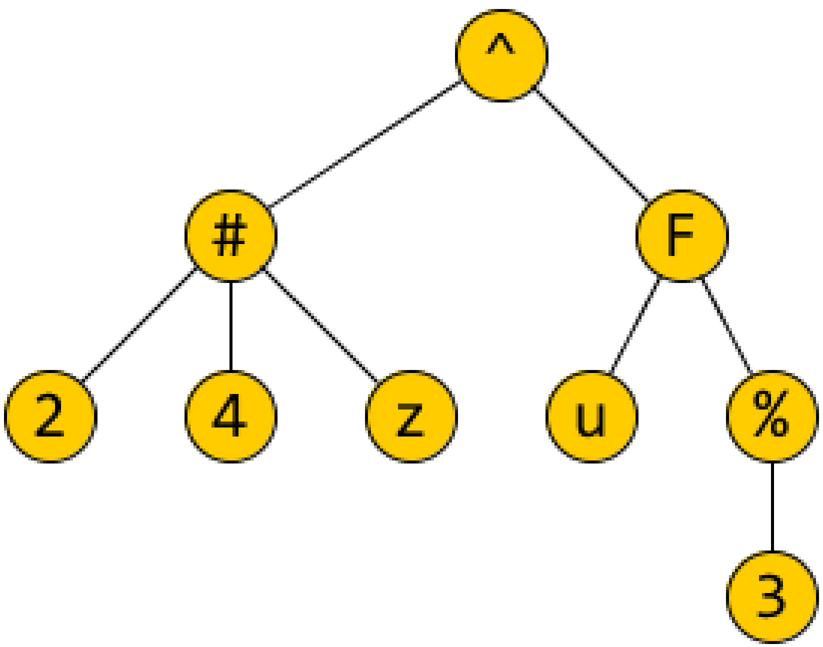
Example: Draw the AST of the following expression, which is written in Lisp notation:

(& 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)



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

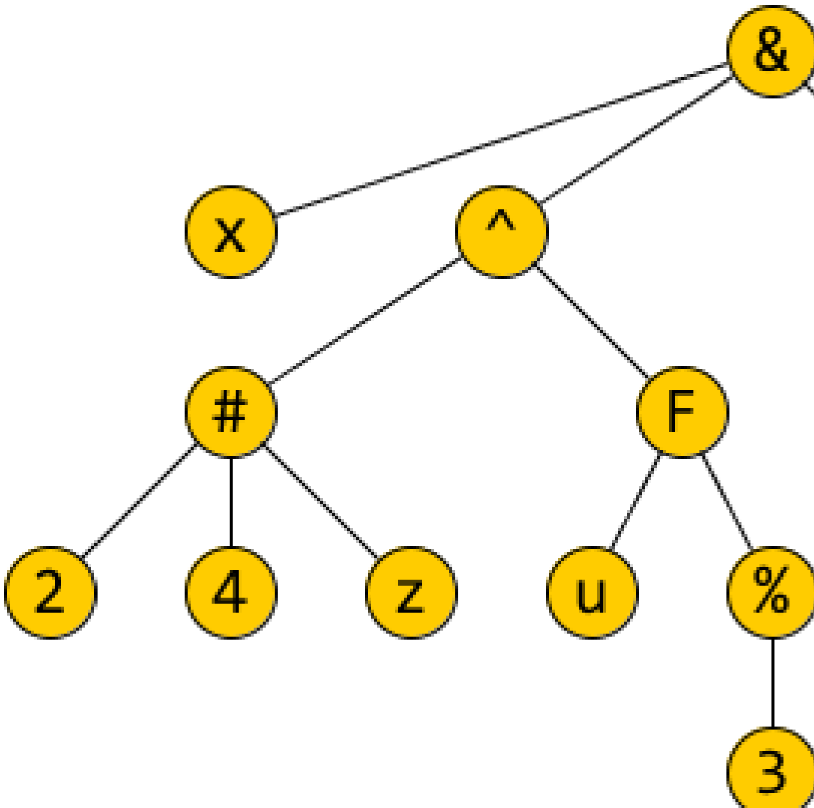
Example: Draw the AST of the following expression, which is written in Lisp notation:

(& 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)



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

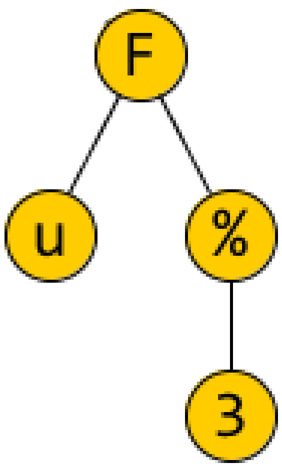
Example: Draw the AST of the following expression, which is written in "rpnLisp" notation:

(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 &)



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

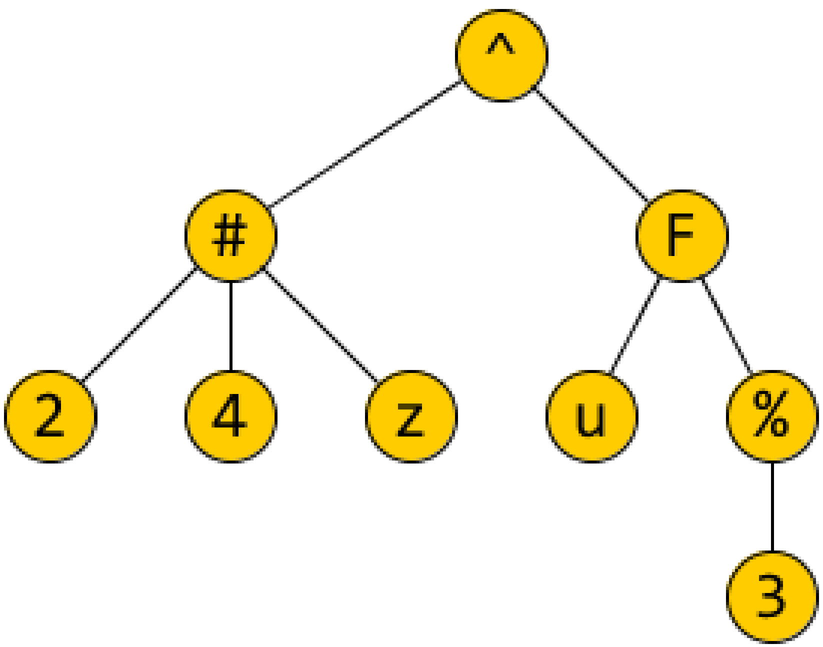
Example: Draw the AST of the following expression, which is written in "rpnLisp" notation:

(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 &)



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

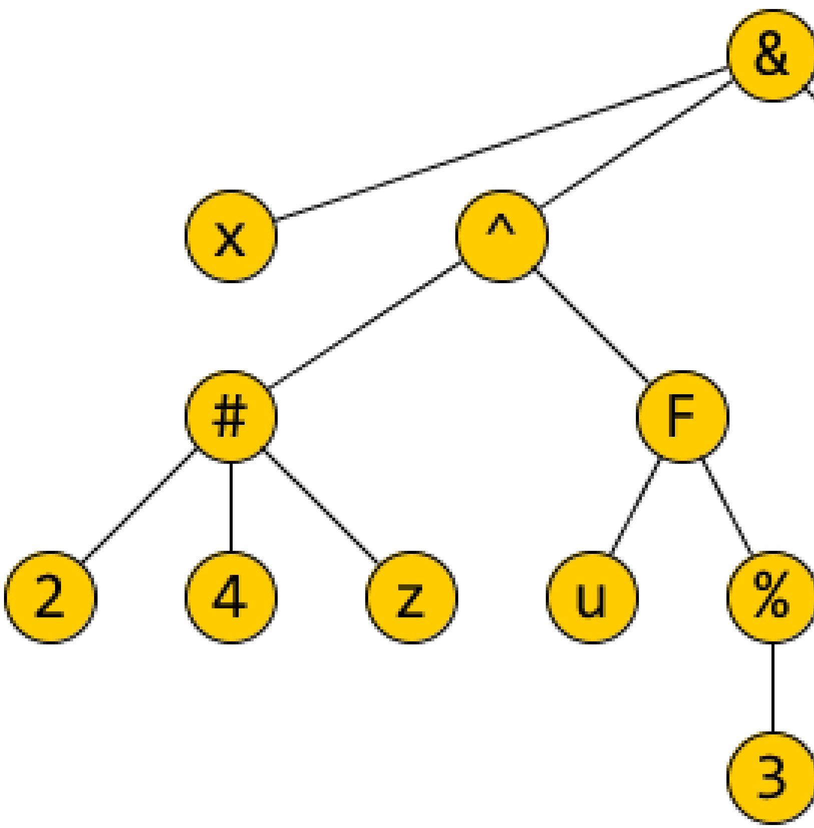
Example: Draw the AST of the following expression, which is written in “rpnLisp” notation:

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

Solution:

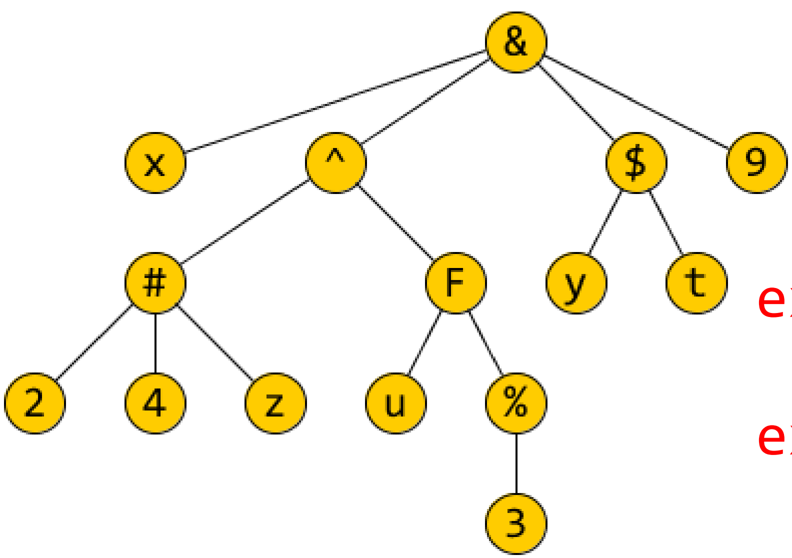
This is the AST of:

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



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

- 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 …

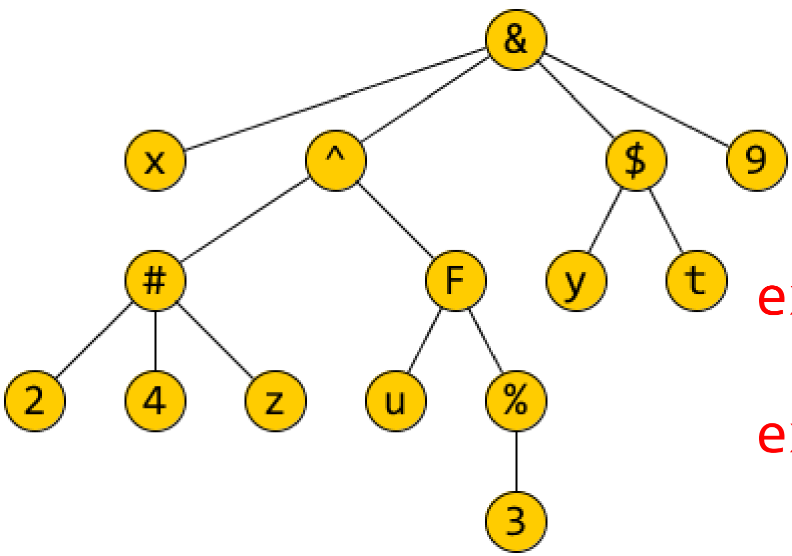


expression in ***prefix*** notation:

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

expression in ***postfix*** notation:

- 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.



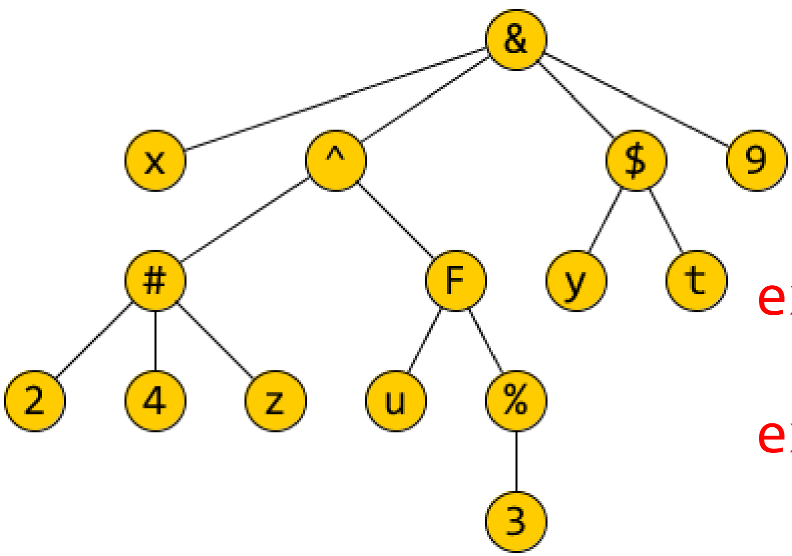
expression in ***prefix*** notation:

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

expression in ***postfix*** notation:

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

- 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.



We can translate an *infix* expression into *prefix* or *postfix* notation by drawing its AST and doing preorder or postorder traversal of the tree.

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 &

Let's translate the infix expr below into prefix & postfix notations

+ 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.

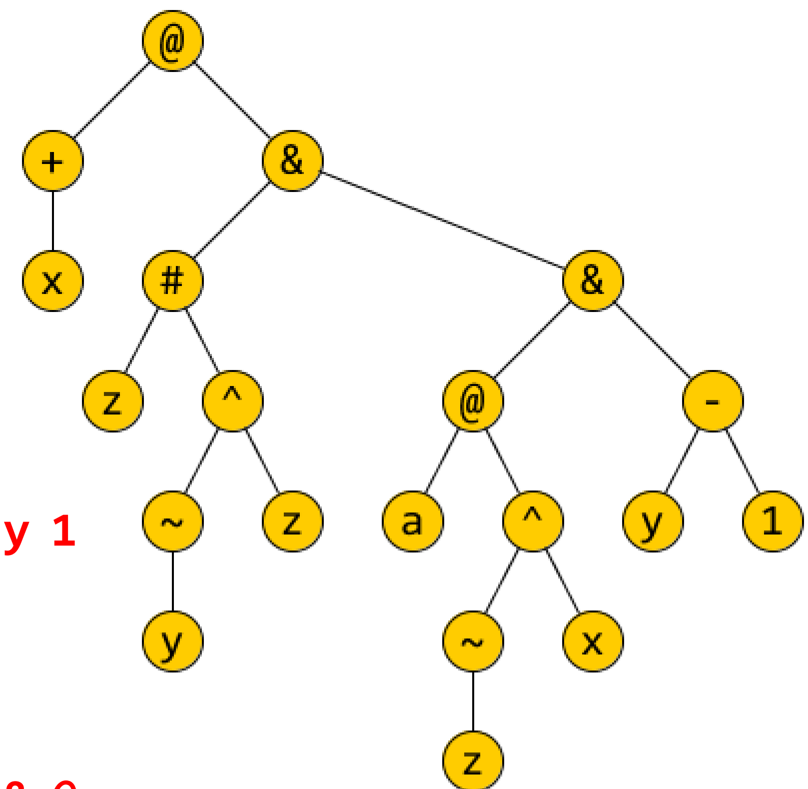
We can translate the above *infix* expression into *prefix* or *postfix* notation by doing preorder or postorder traversal of its AST.

Let’s translate the infix expr below into prefix & postfix notations:

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

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

We can translate the above *infix* expression into *prefix* or *postfix* notation by doing preorder or postorder traversal of its AST.



Equivalent expression in prefix notation:

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

Equivalent expression in postfix notation:

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

Precedence & Associativity Rules ***Don't*** Always Determine the Operator of an Infix Expression That Should be Applied ***First***

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

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

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

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

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

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

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

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

The last homework exercise in section A of the **Syntax-Reading-and-Exercises.pdf** document on Brightspace asks you to evaluate a postfix expression in this way!

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

Evaluation of Postfix Expressions Using a Stack

Postfix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

UNREAD **INPUT: x 2 3 y $+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item):

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

***UNREAD* INPUT:** x **2 3 y $+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item): 7

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

UNREAD **INPUT: 2 3 y $+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

STACK **(rightmost item = topmost item): 7 2**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

***UNREAD* INPUT:** 3 **y $+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item): 7 2 3

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

UNREAD **INPUT:** y **$+_3$ 1 x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item)**: 7 2 3 4**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

***UNREAD* INPUT:** $+_3$ **1 x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item): 7 2 3 4 **9**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

UNREAD **INPUT:** 1 **x $*_2$ $-_2$ 5 $*_3$**

STACK (rightmost item = topmost item): **7 9 1**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

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

STACK (rightmost item = topmost item)**: 7 9 1 7**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

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

STACK (rightmost item = topmost item): **7 9 1 7 7**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

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

STACK (rightmost item = topmost item): 7 9 7 2

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

***UNREAD* INPUT:** 5 $*_3$

STACK (rightmost item = topmost item): **7 2 5**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

UNREAD **INPUT:** $*_3$

STACK (rightmost item = topmost item): 7 2 5 **70**

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.

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.post.e.

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

value of expression

***UNREAD* INPUT:**

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

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

***UNREAD* INPUT:** $*_3$ x $-_2$ $+_3$ 2 3 y $*_2$ 1 x 5

STACK (**leftmost** item = topmost item):

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

***UNREAD* INPUT:** $*_3$ **x** $-_2$ $+_3$ **2 3 y** $*_2$ **1 x** 5

STACK (leftmost item = topmost item): **5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

UNREAD **INPUT:** $*_3$ **x** $-_2$ $+_3$ **2 3 y** $*_2$ **1** x

STACK **(leftmost item = topmost item):** **7 5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

***UNREAD* INPUT:** $*_3$ **x** $-_2$ $+_3$ **2 3 y** $*_2$ 1

STACK (**leftmost** **item = topmost item**): **1 7 5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **7** 1 7 **5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **4 7 5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item)**:** **3 4 7 5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): 2 3 4 7 5

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **9** 2 3 4 **7** **5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **2** 9 7 **5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **7 2 5**

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the only thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

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

STACK (leftmost item = topmost item): **70** 7 2 5

Evaluation of Prefix Expressions Using a Stack

Prefix expressions can be evaluated as follows:

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

When the whole expression has been processed in this way, its value will be the <u>only</u> thing on the stack: This can be proved by induction using the definition of an s.v.pre.e.

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

***UNREAD* INPUT:**

STACK (leftmost item = topmost item): **70** ← value of expression

Translating Prefix/Postfix Notations to Lisp/"rpnLisp"

Recall:

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

NOTE: Translating a prefix/postfix expression into a Lisp/rpnLisp expression *can be the first step in constructing the **abstract syntax tree** of the prefix/postfix expression*, because it's easy to draw a Lisp/rpnLisp expression's AST!

Translating a prefix/postfix expression into Lisp/rpnLisp *can also be the first step in translating prefix notation into postfix or vice versa*, because it's very easy to translate Lisp into rpnLisp or vice versa!

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 **op** $e_1 \ldots e_k$ and $e_1 \ldots e_k$ **op** for the Lisp and rpnLisp expressions (**op** $e_1 \ldots e_k$) and ($e_1 \ldots e_k$ **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 **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 <u>insert parentheses</u> 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**).

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

will be written **[$*_3$ x [$-_2$ [$+_3$ 2 3 y] [$*_2$ w x]] 5]**

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

will be written **[x [[2 3 y $+_3$] [w x $*_2$] $-_2$] 5 $*_3$]**

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 e_1 ... e_k **op**.

e_i is ith-last expression to be popped, and is the ith operand of 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 $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $e_1 \ldots e_k$ **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:** $x \ 2 \ 3 \ +_3 \ y \ -_2 \ u \ x \ 5 \ *_2 \ *_3 \ -_1$

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, ..., e_1.
  - *Push* the rpnLisp expr e_1 ... e_k **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 **2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

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, ..., e_1.
  - *Push* the rpnLisp expr e_1 ... e_k **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 e_1 ... e_k **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 **$+_3$ 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 $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $e_1 \ldots e_k$ **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: **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 $e_1 \ldots e_k$ **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:** $y \; -_2 \; u \; x \; 5 \; *_2 \; *_3 \; -_1$

STACK: $x \; 2 \; 3 \; +_3$ 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 $e_1 \ldots e_k$ **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:** $-_2$ **u x 5 $*_2$ $*_3$ $-_1$**

STACK: [[x 2 3 $+_3$] y $-_2$]

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 e_1 ... e_k **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:** u **x 5 $*_2$ $*_3$ $-_1$**

STACK: **x 2 3 $+_3$ y $-_2$ 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 $e_1 \ldots e_k$ **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: $x\ 2\ 3\ +_3\ y\ -_2$ 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, ..., e_1.
  - *Push* the rpnLisp expr e_1 ... e_k **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:** **5 $*_2$ $*_3$ $-_1$**

STACK: **x 2 3 $+_3$ y $-_2$ u x 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 e_1 ... e_k **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:** $*_2$ $*_3$ $-_1$

STACK: [[x 2 3 $+_3$] y $-_2$] u [x 5 $*_2$]

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 e_1 ... e_k **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:** $*_3$ $-_1$

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

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 e_1 ... e_k **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:** $-_1$

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

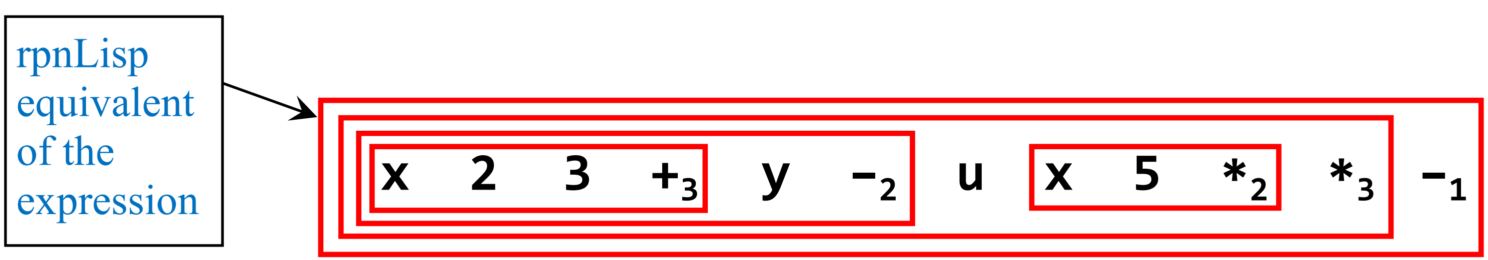
We can use a stack as follows to translate a postfix expression to "rpnLisp":

- 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, ..., e_1.
  - *Push* the rpnLisp expr e_1 ... e_k **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.



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 **op** e_1 ... e_k .

e_i is i^{th} expression to be popped, and is the i^{th} operand of op.

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 **op** e_1 ... 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, ..., e_k.
  - *Push* the Lisp expression **op** e_1 ... 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 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 $e_1, \ldots, e_k$.
  - *Push* the Lisp expression **(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 **op** e_1 ... 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:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2$ 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 (**op** e_1 ... 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 $*_2$

STACK: ($*_2$ 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 (**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

STACK: y $*_2$ 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, ..., e_k.
  - *Push* the Lisp expression **(op** e_1 ... 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 ($*_2$ 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, ..., e_k.
  - *Push* the Lisp expression (**op** e_1 ... 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

STACK: 2 3 y ($*_2$ 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, ..., e_k.
  - *Push* the Lisp expression [**op** e_1 ... 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 $-_2$ $+_3$

STACK: [$+_3$ 2 3 y] [$*_2$ 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, ..., e_k.
  - *Push* the Lisp expression **op** e_1 ... 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 $-_2$

STACK: $-_2$ $+_3$ 2 3 y $*_2$ 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, ..., e_k.
  - *Push* the Lisp expression **op** e_1 ... 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

STACK: x $-_2$ $+_3$ 2 3 y $*_2$ 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, ..., e_k.
  - *Push* the Lisp expression **(op** e_1 ... 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:** $*_3$

STACK: [$*_3$ x [$-_2$ [$+_3$ 2 3 y] [$*_2$ 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, ..., e_k.
    - *Push* the Lisp expression [**op** e_1 ... 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: [$*_3$ x [$-_2$ [$+_3$ 2 3 y] [$*_2$ 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.

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: Substituting $@_3$, $!_3$, $\hat{}_2$, $\#_2$, and $\sim_1$ for $+_3$, $*_3$, $*_2$, $-_2$, and $-_1$ in our solution to the latter problem gives a solution to the former problem.

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.

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: