

# Logical operators full truth table

Input		Output				
$p$	$q$	Conjunction $p \wedge q$	Exclusive or $p \oplus q$	Disjunction $p \vee q$	Conditional $p \rightarrow q$	Biconditional $p \leftrightarrow q$
$T$	$T$	$T$	$F$	$T$	$T$	$T$
$T$	$F$	$F$	$T$	$T$	$F$	$F$
$F$	$T$	$F$	$T$	$T$	$T$	$F$
$F$	$F$	$F$	$F$	$F$	$T$	$T$
		" $p$ and $q$ "	" $p$ xor $q$ "	" $p$ or $q$ "	"if $p$ then $q$ "	" $p$ if and only if $q$ "

## Hypothesis conclusion

The only way to make the conditional statement  $p \rightarrow q$  false is to \_\_\_\_\_

The **hypothesis** of  $p \rightarrow q$  is \_\_\_\_\_ The **antecedent** of  $p \rightarrow q$  is \_\_\_\_\_

The **conclusion** of  $p \rightarrow q$  is \_\_\_\_\_ The **consequent** of  $p \rightarrow q$  is \_\_\_\_\_

## Converse inverse contrapositive

The **converse** of  $p \rightarrow q$  is \_\_\_\_\_

The **inverse** of  $p \rightarrow q$  is \_\_\_\_\_

The **contrapositive** of  $p \rightarrow q$  is \_\_\_\_\_

# Compound propositions recursive definition

We can use a recursive definition to describe all compound propositions that use propositional variables from a specified collection. Here's the definition for all compound propositions whose propositional variables are in  $\{p, q\}$ .

Basis Step:  $p$  and  $q$  are each a compound proposition  
Recursive Step: If  $x$  is a compound proposition then so is  $(\neg x)$  and if  $x$  and  $y$  are both compound propositions then so is each of  $(x \wedge y), (x \oplus y), (x \vee y), (x \rightarrow y), (x \leftrightarrow y)$

# Compound propositions precedence

Order of operations (Precedence) for logical operators:

Negation, then conjunction / disjunction, then conditional / biconditionals.

Example:  $\neg p \vee \neg q$  means  $(\neg p) \vee (\neg q)$ .

# Logical equivalence identities

## (Some) logical equivalences

*Can replace  $p$  and  $q$  with any compound proposition*

$$\neg(\neg p) \equiv p$$

**Double negation**

$$p \vee q \equiv q \vee p$$

$$p \wedge q \equiv q \wedge p$$

**Commutativity** Ordering of terms

$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

**Associativity** Grouping of terms

$$p \wedge F \equiv F$$

$$p \vee T \equiv T$$

$$p \wedge T \equiv p$$

$$p \vee F \equiv p$$

**Domination** aka short circuit evaluation

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

**DeMorgan's Laws**

$$p \rightarrow q \equiv \neg p \vee q$$

$$p \rightarrow q \equiv \neg q \rightarrow \neg p$$

**Contrapositive**

$$\neg(p \rightarrow q) \equiv p \wedge \neg q$$

$$\neg(p \leftrightarrow q) \equiv p \oplus q$$

$$p \leftrightarrow q \equiv q \leftrightarrow p$$

*Extra examples:*

$p \leftrightarrow q$  is not logically equivalent to  $p \wedge q$  because \_\_\_\_\_

$p \rightarrow q$  is not logically equivalent to  $q \rightarrow p$  because \_\_\_\_\_

# Logical operators english synonyms

## Common ways to express logical operators in English:

**Negation**  $\neg p$  can be said in English as

- Not  $p$ .
- It's not the case that  $p$ .
- $p$  is false.

**Conjunction**  $p \wedge q$  can be said in English as

- $p$  and  $q$ .
- Both  $p$  and  $q$  are true.
- $p$  but  $q$ .

**Exclusive or**  $p \oplus q$  can be said in English as

- $p$  or  $q$ , but not both.
- Exactly one of  $p$  and  $q$  is true.

**Disjunction**  $p \vee q$  can be said in English as

- $p$  or  $q$ , or both.
- $p$  or  $q$  (inclusive).
- At least one of  $p$  and  $q$  is true.

**Conditional**  $p \rightarrow q$  can be said in English as

- |                               |                               |
|-------------------------------|-------------------------------|
| • if $p$ , then $q$ .         | • $q$ follows from $p$ .      |
| • $p$ is sufficient for $q$ . | • $p$ is sufficient for $q$ . |
| • $q$ when $p$ .              | • $q$ is necessary for $p$ .  |
| • $q$ whenever $p$ .          | • $p$ only if $q$ .           |
| • $p$ implies $q$ .           |                               |

**Biconditional**

- $p$  if and only if  $q$ .
- $p$  iff  $q$ .
- If  $p$  then  $q$ , and conversely.
- $p$  is necessary and sufficient for  $q$ .



## Predicate examples finite domain

Input $x$	Output		
	$V(x)$ $[x]_{2c,3} > 0$	$N(x)$ $[x]_{2c,3} < 0$	$Mystery(x)$
000	$F$		$T$
001	$T$		$T$
010	$T$		$T$
011	$T$		$F$
100	$F$		$F$
101	$F$		$T$
110	$F$		$F$
111	$F$		$T$

The domain for each of the predicates  $V(x)$ ,  $N(x)$ ,  $Mystery(x)$  is \_\_\_\_\_.

Fill in the table of values for the predicate  $N(x)$  based on the formula given.

## Predicate truth set definition

**Definition:** The **truth set** of a predicate is the collection of all elements in its domain where the predicate evaluates to  $T$ .

Notice that specifying the domain and the truth set is sufficient for defining a predicate.

## Predicate truth set example

The truth set for the predicate  $V(x)$  is \_\_\_\_\_.

The truth set for the predicate  $N(x)$  is \_\_\_\_\_.

The truth set for the predicate  $Mystery(x)$  is \_\_\_\_\_.

# Quantification definition

The **universal quantification** of predicate  $P(x)$  over domain  $U$  is the statement “ $P(x)$  for all values of  $x$  in the domain  $U$ ” and is written  $\forall x P(x)$  or  $\forall x \in U P(x)$ . When the domain is finite, universal quantification over the domain is equivalent to iterated *conjunction* (ands).

The **existential quantification** of predicate  $P(x)$  over domain  $U$  is the statement “There exists an element  $x$  in the domain  $U$  such that  $P(x)$ ” and is written  $\exists x P(x)$  for  $\exists x \in U P(x)$ . When the domain is finite, existential quantification over the domain is equivalent to iterated *disjunction* (ors).

An element for which  $P(x) = F$  is called a **counterexample** of  $\forall x P(x)$ .

An element for which  $P(x) = T$  is called a **witness** of  $\exists x P(x)$ .

## Quantification logical equivalence

Statements involving predicates and quantifiers are **logically equivalent** means they have the same truth value no matter which predicates (domains and functions) are substituted in.

**Quantifier version of De Morgan’s laws:**  $\neg \forall x P(x) \equiv \exists x (\neg P(x))$   $\neg \exists x Q(x) \equiv \forall x (\neg Q(x))$

## Quantification examples finite domain

Examples of quantifications using  $V(x), N(x), Mystery(x)$ :

**True or False:**  $\exists x ( V(x) \wedge N(x) )$

**True or False:**  $\forall x ( V(x) \rightarrow N(x) )$

**True or False:**  $\exists x ( N(x) \leftrightarrow Mystery(x) )$

Rewrite  $\neg \forall x ( V(x) \oplus Mystery(x) )$  into a logical equivalent statement.

Notice that these are examples where the predicates have *finite* domain. How would we evaluate quantifications where the domain may be infinite?

# Rna rnaalen basecount definitions

*Recall the definitions:* The set of RNA strands  $S$  is defined (recursively) by:

Basis Step:  $\mathbf{A} \in S, \mathbf{C} \in S, \mathbf{U} \in S, \mathbf{G} \in S$   
Recursive Step: If  $s \in S$  and  $b \in B$ , then  $sb \in S$

where  $sb$  is string concatenation.

The function  $rnalen$  that computes the length of RNA strands in  $S$  is defined recursively by:

$rnalen : S \rightarrow \mathbb{Z}^+$   
Basis Step: If  $b \in B$  then  $rnalen(b) = 1$   
Recursive Step: If  $s \in S$  and  $b \in B$ , then  $rnalen(sb) = 1 + rnalen(s)$

The function  $basecount$  that computes the number of a given base  $b$  appearing in a RNA strand  $s$  is defined recursively by:

$basecount : S \times B \rightarrow \mathbb{N}$   
Basis Step: If  $b_1 \in B, b_2 \in B$   $basecount( (b_1, b_2) ) = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases}$   
Recursive Step: If  $s \in S, b_1 \in B, b_2 \in B$   $basecount( (sb_1, b_2) ) = \begin{cases} 1 + basecount( (s, b_2) ) & \text{when } b_1 = b_2 \\ basecount( (s, b_2) ) & \text{when } b_1 \neq b_2 \end{cases}$

## Predicate rna example

**Example predicates on  $S$ , the set of RNA strands (an infinite set)**

$H : S \rightarrow \{T, F\}$  where  $H(s) = T$  for all  $s$ .

Truth set of  $H$  is \_\_\_\_\_

$F_A : S \rightarrow \{T, F\}$  defined recursively by:

Basis step:  $F_A(\mathbf{A}) = T, F_A(\mathbf{C}) = F_A(\mathbf{G}) = F_A(\mathbf{U}) = F$

Recursive step: If  $s \in S$  and  $b \in B$ , then  $F_A(sb) = F_A(s)$ .

Example where  $F_A$  evaluates to  $T$  is \_\_\_\_\_

Example where  $F_A$  evaluates to  $F$  is \_\_\_\_\_



# Predicates example *rlen* basecount

Using functions to define predicates:

$L$  with domain  $S \times \mathbb{Z}^+$  is defined by, for  $s \in S$  and  $n \in \mathbb{Z}^+$ ,

$$L( (s, n) ) = \begin{cases} T & \text{if } rlen(s) = n \\ F & \text{otherwise} \end{cases}$$

In other words,  $L( (s, n) )$  means  $rlen(s) = n$

$BC$  with domain  $S \times B \times \mathbb{N}$  is defined by, for  $s \in S$  and  $b \in B$  and  $n \in \mathbb{N}$ ,

$$BC( (s, b, n) ) = \begin{cases} T & \text{if } basecount( (s, b) ) = n \\ F & \text{otherwise} \end{cases}$$

In other words,  $BC( (s, b, n) )$  means  $basecount( (s, b) ) = n$

Example where  $L$  evaluates to  $T$ : \_\_\_\_\_ Why?

Example where  $BC$  evaluates to  $T$ : \_\_\_\_\_ Why?

Example where  $L$  evaluates to  $F$ : \_\_\_\_\_ Why?

Example where  $BC$  evaluates to  $F$ : \_\_\_\_\_ Why?

$$\exists t \ BC(t) \qquad \exists (s, b, n) \in S \times B \times \mathbb{N} \ (basecount( (s, b) ) = n)$$

In English:

Witness that proves this existential quantification is true:

$$\forall t \ BC(t) \qquad \forall (s, b, n) \in S \times B \times \mathbb{N} \ (basecount( (s, b) ) = n)$$

In English:

Counterexample that proves this universal quantification is false:

# Predicates projecting example rna basecount

## New predicates from old

1. Define the **new** predicate with domain  $S \times B$  and rule

$$\text{basecount}( (s, b) ) = 3$$

Example domain element where predicate is  $T$ :

2. Define the **new** predicate with domain  $S \times \mathbb{N}$  and rule

$$\text{basecount}( (s, \mathbf{A}) ) = n$$

Example domain element where predicate is  $T$ :

3. Define the **new** predicate with domain  $S \times B$  and rule

$$\exists n \in \mathbb{N} (\text{basecount}( (s, b) ) = n)$$

Example domain element where predicate is  $T$ :

4. Define the **new** predicate with domain  $S$  and rule

$$\forall b \in B (\text{basecount}( (s, b) ) = 1)$$

Example domain element where predicate is  $T$ :

## Predicate notation

**Notation:** for a predicate  $P$  with domain  $X_1 \times \cdots \times X_n$  and a  $n$ -tuple  $(x_1, \dots, x_n)$  with each  $x_i \in X$ , we can write  $P(x_1, \dots, x_n)$  to mean  $P( (x_1, \dots, x_n) )$ .

# Nested quantifiers

## Nested quantifiers

$$\forall s \in S \forall b \in B \forall n \in \mathbb{N} (\text{basecount}(s, b) = n)$$

In English:

Counterexample that proves this universal quantification is false:

$$\forall n \in \mathbb{N} \forall s \in S \forall b \in B (\text{basecount}(s, b) = n)$$

In English:

Counterexample that proves this universal quantification is false:

## Compound proposition definitions

**Proposition:** Declarative sentence that is true or false (not both).

**Propositional variable:** Variable that represents a proposition.

**Compound proposition:** New proposition formed from existing propositions (potentially) using logical operators. *Note:* A propositional variable is one example of a compound proposition.

**Truth table:** Table with one row for each of the possible combinations of truth values of the input and an additional column that shows the truth value of the result of the operation corresponding to a particular row.

# Logical equivalence

**Logical equivalence** : Two compound propositions are **logically equivalent** means that they have the same truth values for all settings of truth values to their propositional variables.

**Tautology**: A compound proposition that evaluates to true for all settings of truth values to its propositional variables; it is abbreviated  $T$ .

**Contradiction**: A compound proposition that evaluates to false for all settings of truth values to its propositional variables; it is abbreviated  $F$ .

**Contingency**: A compound proposition that is neither a tautology nor a contradiction.

## Tautology contradiction contingency examples

Label each of the following as a tautology, contradiction, or contingency.

$p \wedge p$

$p \oplus p$

$p \vee p$

$p \vee \neg p$

$p \wedge \neg p$

## Logical equivalence extra example

*Extra Example:* Which of the compound propositions in the table below are logically equivalent?

Input		Output				
$p$	$q$	$\neg(p \wedge \neg q)$	$\neg(\neg p \vee \neg q)$	$(\neg p \vee q)$	$(\neg q \vee \neg p)$	$(p \wedge q)$
$T$	$T$					
$T$	$F$					
$F$	$T$					
$F$	$F$					

# Algorithm definition

**New!** An algorithm is a finite sequence of precise instructions for solving a problem.

Algorithms can be expressed in English or in more formalized descriptions like pseudocode or fully executable programs.

Sometimes, we can define algorithms whose output matches the rule for a function we already care about. Consider the (integer) logarithm function

$$\text{log}_b : \{b \in \mathbb{Z} \mid b > 1\} \times \mathbb{Z}^+ \rightarrow \mathbb{N}$$

defined by

$$\text{log}_b( (b,n) ) = \text{greatest integer } y \text{ so that } b^y \text{ is less than or equal to } n$$

Calculating integer part of base  $b$  logarithm

```
1  procedure logb( $b,n$ : positive integers with  $b > 1$ )
2     $i := 0$ 
3    while  $n > b - 1$ 
4       $i := i + 1$ 
5       $n := n \text{ div } b$ 
6    return  $i$  { $i$  holds the integer part of the base  $b$  logarithm of  $n$ }
```

Trace this algorithm with inputs  $b = 3$  and  $n = 17$

	$b$	$n$	$i$	$n > b - 1?$
Initial value	3	17		
After 1 iteration				
After 2 iterations				
After 3 iterations				

Compare: does the output match the rule for the (integer) logarithm function?

# Fixed width definition

**Definition** For  $b$  an integer greater than 1,  $w$  a positive integer, and  $n$  a nonnegative integer \_\_\_\_\_, the **base  $b$  fixed-width  $w$  expansion of  $n$**  is

$$(a_{w-1} \cdots a_1 a_0)_{b,w}$$

where  $a_0, a_1, \dots, a_{w-1}$  are nonnegative integers less than  $b$  and

$$n = \sum_{i=0}^{w-1} a_i b^i$$

# Fixed width example

Decimal $b = 10$	Binary $b = 2$	Binary fixed-width 10 $b = 2, w = 10$	Binary fixed-width 7 $b = 2, w = 7$	Binary fixed-width 4 $b = 2, w = 4$
$(20)_{10}$				

# Fixed width fractional definition

**Definition** For  $b$  an integer greater than 1,  $w$  a positive integer,  $w'$  a positive integer, and  $x$  a real number the **base  $b$  fixed-width expansion of  $x$  with integer part width  $w$  and fractional part width  $w'$**  is  $(a_{w-1} \cdots a_1 a_0 . c_1 \cdots c_{w'})_{b,w,w'}$  where  $a_0, a_1, \dots, a_{w-1}, c_1, \dots, c_{w'}$  are nonnegative integers less than  $b$  and

$$x \geq \sum_{i=0}^{w-1} a_i b^i + \sum_{j=1}^{w'} c_j b^{-j} \quad \text{and} \quad x < \sum_{i=0}^{w-1} a_i b^i + \sum_{j=1}^{w'} c_j b^{-j} + b^{-w'}$$

3.75 in fixed-width binary, integer part width 2, fractional part width 8	
0.1 in fixed-width binary, integer part width 2, fractional part width 8	

```
welcome $jshell
| Welcome to JShell -- Version 10.0.1
| For an introduction type: /help intro

[jshell> 0.1
$1 ==>

[jshell> 0.2
$2 ==>

[jshell> 0.1 + 0.2
$3 ==>

[jshell> Math.sqrt(2)
$4 ==>

[jshell> Math.sqrt(2)*Math.sqrt(2)
$5 ==>

[jshell> █
```

Note: Java uses floating point, not fixed width representation, but similar rounding errors appear in both.

# Negative int expansions

**Representing negative integers in binary:** Fix a positive integer width for the representation  $w$ ,  $w > 1$ .

	To represent a positive integer $n$	To represent a negative integer $-n$
Sign-magnitude	$[0a_{w-2} \cdots a_0]_{s,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$  Example $n = 17$ , $w = 7$ :	$[1a_{w-2} \cdots a_0]_{s,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$  Example $-n = -17$ , $w = 7$ :
2s complement	$[0a_{w-2} \cdots a_0]_{2c,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$  Example $n = 17$ , $w = 7$ :	$[1a_{w-2} \cdots a_0]_{2c,w}$ , where $2^{w-1} - n = (a_{w-2} \cdots a_0)_{2,w-1}$  Example $-n = -17$ , $w = 7$ :

## Calculating 2s complement

For positive integer  $n$ , to represent  $-n$  in 2s complement with width  $w$ ,

- Calculate  $2^{w-1} - n$ , convert result to binary fixed-width  $w - 1$ , pad with leading 1, or
- Express  $-n$  as a sum of powers of 2, where the leftmost  $2^{w-1}$  is negative weight, or
- Convert  $n$  to binary fixed-width  $w$ , flip bits, add 1 (ignore overflow)

*Challenge: use definitions to explain why each of these approaches works.*



# Representing zero

## Representing 0:

So far, we have representations for positive and negative integers. What about 0?

	To represent a <b>non-negative</b> integer $n$	To represent a <b>non-positive</b> integer $-n$
Sign-magnitude	$[0a_{w-2} \cdots a_0]_{s,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 0$ , $w = 7$ :	$[1a_{w-2} \cdots a_0]_{s,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0$ , $w = 7$ :
2s complement	$[0a_{w-2} \cdots a_0]_{2c,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 0$ , $w = 7$ :	$[1a_{w-2} \cdots a_0]_{2c,w}$ , where $2^{w-1} - n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0$ , $w = 7$ :

## Netflix intro

What data should we encode about each Netflix account holder to help us make effective recommendations?

In machine learning, clustering can be used to group similar data for prediction and recommendation. For example, each Netflix user's viewing history can be represented as a  $n$ -tuple indicating their preferences about movies in the database, where  $n$  is the number of movies in the database. People with similar tastes in movies can then be clustered to provide recommendations of movies for one another. Mathematically, clustering is based on a notion of distance between pairs of  $n$ -tuples.

# Data types

Term	Examples: (add additional examples from class)
<b>set</b> unordered collection of elements <i>repetition doesn't matter</i> <i>Equal sets agree on membership of all elements</i>	$7 \in \{43, 7, 9\}$ $2 \notin \{43, 7, 9\}$
<b><math>n</math>-tuple</b> ordered sequence of elements with $n$ “slots” ( $n > 0$ ) <i>repetition matters, fixed length</i> <i>Equal <math>n</math>-tuples have corresponding components equal</i>	
<b>string</b> ordered finite sequence of elements each from specified set (called the alphabet over which the string is defined) <i>repetition matters, arbitrary finite length</i> <i>Equal strings have same length and corresponding characters equal</i>	

*Special cases:*

When  $n = 2$ , the 2-tuple is called an **ordered pair**.

A string of length 0 is called the **empty string** and is denoted  $\lambda$ .

A set with no elements is called the **empty set** and is denoted  $\{\}$  or  $\emptyset$ .

# Set operations

To define a set we can use the roster method, set builder notation, a recursive definition, and also we can apply a set operation to other sets.

## New! Cartesian product of sets and set-wise concatenation of sets of strings

**Definition:** Let  $X$  and  $Y$  be sets. The **Cartesian product** of  $X$  and  $Y$ , denoted  $X \times Y$ , is the set of all ordered pairs  $(x, y)$  where  $x \in X$  and  $y \in Y$

$$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$$

Conventions: (1) Cartesian products can be chained together to result in sets of  $n$ -tuples and (2) When we form the Cartesian product of a set with itself  $X \times X$  we can denote that set as  $X^2$ , or  $X^n$  for the Cartesian product of a set with itself  $n$  times for a positive integer  $n$ .

**Definition:** Let  $X$  and  $Y$  be sets of strings over the same alphabet. The **set-wise concatenation** of  $X$  and  $Y$ , denoted  $X \circ Y$ , is the set of all results of string concatenation  $xy$  where  $x \in X$  and  $y \in Y$

$$X \circ Y = \{xy \mid x \in X \text{ and } y \in Y\}$$

**Pro-tip:** the meaning of writing one element next to another like  $xy$  depends on the data-types of  $x$  and  $y$ . When  $x$  and  $y$  are strings, the convention is that  $xy$  is the result of string concatenation. When  $x$  and  $y$  are numbers, the convention is that  $xy$  is the result of multiplication. This is (one of the many reasons) why is it very important to declare the data-type of variables before we use them.

*Fill in the missing entries in the table:*

Set	Example elements in this set and their data type:			
$B$	A	C	G	U
	(A, C)		(U, U)	
$B \times \{-1, 0, 1\}$				
$\{-1, 0, 1\} \times B$				
	(0, 0, 0)			
$\{A, C, G, U\} \circ \{A, C, G, U\}$				
	GGGG			

# Defining functions

**New! Defining functions** A function is defined by its (1) domain, (2) codomain, and (3) rule assigning each element in the domain exactly one element in the codomain.

The domain and codomain are nonempty sets.

The rule can be depicted as a table, formula, piecewise definition, or English description.

The notation is

“Let the function FUNCTION-NAME: DOMAIN  $\rightarrow$  CODOMAIN be given by  
FUNCTION-NAME( $x$ ) = ...for every  $x \in DOMAIN$ ”.

or

“Consider the function FUNCTION-NAME: DOMAIN  $\rightarrow$  CODOMAIN defined as  
FUNCTION-NAME( $x$ ) = ...for every  $x \in DOMAIN$ ”.

Example: The absolute value function

**Domain**

**Codomain**

**Rule**

## Defining functions recursively

When the domain of a function is a *recursively defined set*, the rule assigning images to domain elements (outputs) can also be defined recursively.

Recall: The set of RNA strands  $S$  is defined (recursively) by:

$$\begin{array}{ll} \text{Basis Step:} & \mathbf{A} \in S, \mathbf{C} \in S, \mathbf{U} \in S, \mathbf{G} \in S \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B, \text{ then } sb \in S \end{array}$$

where  $sb$  is string concatenation.

**Definition** (Of a function, recursively) A function  $rnalen$  that computes the length of RNA strands in  $S$  is defined by:

$$\begin{array}{lll} & & rnalen : S \rightarrow \mathbb{Z}^+ \\ \text{Basis Step:} & \text{If } b \in B \text{ then} & rnalen(b) = 1 \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B, \text{ then} & rnalen(sb) = 1 + rnalen(s) \end{array}$$

The domain of  $rnalen$  is

The codomain of  $rnalen$  is

Example function application:

$$rnalen(\mathbf{ACU}) =$$

*Example:* A function  $basecount$  that computes the number of a given base  $b$  appearing in a RNA strand  $s$  is defined recursively:

$$\begin{array}{lll} & & basecount : S \times B \rightarrow \mathbb{N} \\ \text{Basis Step:} & \text{If } b_1 \in B, b_2 \in B & basecount((b_1, b_2)) = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases} \\ \text{Recursive Step:} & \text{If } s \in S, b_1 \in B, b_2 \in B & basecount((sb_1, b_2)) = \begin{cases} 1 + basecount((s, b_2)) & \text{when } b_1 = b_2 \\ basecount((s, b_2)) & \text{when } b_1 \neq b_2 \end{cases} \end{array}$$