disco

Release 1.0

Jan 30, 2023

Contents

CHAPTER 1

Disco Language Reference

The Disco Language Reference consists of a number of short, interlinked pages, each centered on a single topic. This is not a typical "language reference", which would be intended for language implementers and organized according to syntax, type system, semantics, and so on. Neither is it a comprehensive tutorial meant to be read from start to finish. Rather, it is intended to be a reference for students who are just learning the language. Error messages and other documentation may link here, giving students the opportunity to learn about specific topics just when it is relevant to them.

1.1 Arithmetic

Disco can do many of the usual operations on numbers. For more on the types of numbers Disco supports, see *Numeric types*.

1.1.1 Addition

Note: This page concerns the + operator on numbers; for the + operator on *types*, see sum types; for the + operator on graphs, see overlay.

Values of all *numeric types* can be added using the + operator. For example:

```
Disco> 1 + 2
3
Disco> (-5) + 2/3
-13/3
```

If you ask for the *type* of +, you get

```
Disco> :type +
~+~ :  ×  →
```

which says that it is a *function* taking a pair of natural numbers and returning a natural number. However, as mentioned above, it also works on other numeric types such as integers and rational numbers.

1.1.2 Multiplication

Note: This page concerns the * operator on numbers; for the * operator on *types*, see *pair types*; for the * operator on graphs, see connect.

All *numeric types* can be multiplied using the * operator. For example:

```
Disco> 2 * 3
6
Disco> (-5) * 2/3
-10/3
```

In some cases, the * symbol is not needed: putting two things next to each other means multiplying them. For example:

```
Disco> (1 + 2)(3 + 4)
21
Disco> x : N
Disco> x = 5
Disco> 3x
15
```

The exception is that putting a *variable* directly on the left of something else is interpreted as *function application* instead of multiplication.

1.1.3 Subtraction

Integers and *rational numbers* can be subtracted using the − operator. For example:

```
Disco> 1 - 5
-4
Disco> 2/3 - 1/2
1/6
```

It is not possible to subtract *natural numbers* or *fractional numbers* since those *numeric types* have no concept of negative numbers; however, in many situations, if you subtract natural or fractional numbers they will be automatically converted to integers or rationals as appropriate.

In some situations, you really do need to subtract natural or fractional numbers. For example, consider this incorrect definition of the *factorial* function:

```
fact_bad : N -> N
fact_bad(0) = 1
fact_bad(n) = n * fact_bad(n-1)
```

This definition will yield an error, because subtracting two natural numbers is not allowed: in particular, `n` is a natural number and we are attempting to perform the subtraction `n-1`. One solution is to use the special "dot minus" operator `.-` (also written), which simply stops at zero instead of yielding negative numbers:

```
Disco> 5 .- 3
2
Disco> 5 .- 4
1
Disco> 5 .- 5
0
Disco> 5 .- 6
0
Disco> 5 .- 7
0
```

Using dot minus, we can write a correct definition of factorial as follows:

```
fact : N -> N
fact(0) = 1
fact(n) = n * fact(n .- 1)
```

(Alternatively, we could define `fact` using an *arithmetic pattern*; see that page for more info.)

1.1.4 Division

Fractional and *rational numbers* can be divided using the `/` operator. For example:

```
Disco> 1 / 5
1/5
Disco> (2/3) / (7/5)
10/21
```

It is not possible to divide *natural numbers* or *integers* since those *numeric types* have no concept of fractions; however, in many situations, if you divide natural numbers or integers they will be automatically converted to fractionals or rationals as appropriate:

```
Disco> :type 1 * (-2)
1 * (-2) :
Disco> :type 1 / (-2)
1 / (-2) :
```

Some related operations:

- If you want only the *quotient* when dividing (*i.e.* the integer number of times one thing fits into another, disregarding any remainder), you can use *integer division*.
- If you want the *remainder* instead, you can use *the mod operator*.
- If you just want to test whether one number evenly divides another, you should use the *divides operator*.

1.1.5 Integer division

The *integer division* or *quotient* of two numbers, written `//`, is the result of the division *rounded down* to the nearest integer. Intuitively, you can think of it as the number of times that one thing fits into another, disregarding any *remainder*. For example, `11 // 2` is `5`, because `2` fits into `11` five times (with some left over).

```
Disco> 11 // 2
5
Disco> 6 // 2
```

(continues on next page)

(continued from previous page)

```
3
Disco> 6 // 7
0
Disco> (-7) // 2
-4
```

In fact, `//` is simply defined in terms of regular *division* along with the *floor operation*:

```
x // y = floor (x / y)
```

Although *dividing* two integers using the usual `/` operator does not necessarily result in an integer, using integer division does. In particular, the integer division *operator* can be given the types

```
~//~ :  ×  →
~//~ :  ×  →
```

Formally, the result of `//` is defined in terms of the "Division Algorithm": given a number n and a divisor d, the quotient `n // d` is the unique number q such that $n = qd + r$, where $0 \leq r < d$.

1.1.6 Modulo

The `mod` operator is used to give the *remainder* when one number is *divided by* another.

For example, `11 mod 2` is `1`, because `2` fits into `11` five times, with a remainder of 1; `11 mod 4` is `3`, because dividing `11` by `4` leaves a remainder of `3`.

```
Disco> 11 mod 2
1
Disco> 11 mod 4
3
Disco> 6 mod 2
0
Disco> 6 mod 7
6
Disco> (-7) mod 2
1
```

Formally, the result of `mod` is defined in terms of the "Division Algorithm": given a number n and a positive divisor d, the remainder `n mod d` is the unique number r such that $n = qd + r$, where $0 \leq r < d$ and q is the *quotient*. (For negative divisors, we instead require $d < r \leq 0$.)

1.1.7 Divisibility testing

We can test whether one number evenly divides another using the `divides` operator. In particular, `a divides b` is true if there exists an *integer* `k` such that `b == k*a`. For example:

```
Disco> 3 divides 6
true
Disco> 6 divides 3
false
Disco> 3 divides (-6)
true
Disco> 5 divides 5
true
```

(continues on next page)

(continued from previous page)

```
Disco> 4 divides 6
false
Disco> 0 divides 10
false
Disco> 10 divides 0
true
Disco> 0 divides 0
true
Disco> 1/2 divides 3/2
true
```

1.1.8 Exponentiation

The `^` operator is used to raise one number to the power of another.

```
Disco> 2 ^ 5
32
Disco> 2 ^ 0
1
Disco> 2 ^ (-5)
1/32
Disco> (-3) ^ (-5)
-1/243
```

If the exponent is a *natural number*, the result will have the same type as the base. If the exponent is an *integer*, the result will be *fractional* or *rational*, depending on the type of the base.

Fractional exponents may not be used, as the result could be irrational, and Disco has no way to represent irrational numbers. For example, $2^{1/2} = \sqrt{2}$.

1.1.9 Rounding

Sometimes, when we have a *fractional* or *rational* number, we want to get rid of the fractional part and turn it into an *integer* or *natural number*. This can be done with the `floor` and `ceiling` operators.

- `floor x`, also written ⌊ `x` ⌋, returns the largest integer which is less than or equal to `x`. For example:

  ```
  Disco> floor(1/2)
  0
  Disco> floor(7/2)
  3
  Disco> floor(3)
  3
  Disco> floor(-1/2)
  -1
  ```

  Note: That `floor` always rounds *down*, even for negative numbers. This is how mathematicians think about `floor`, and is the most mathematically elegant definition; however, note that in some other programming languages, `floor` always rounds *towards zero* instead, so *e.g.* `floor(-1/2)` would be `0`.

- Likewise, `ceiling x`, also written ⌈ `x` ⌉, returns the smallest integer which is greater than or equal to `x`. For example:

```
Disco> ceiling(1/2)
1
Disco> ceiling(7/2)
4
Disco> ceiling(3)
3
Disco> ceiling(-1/2)
0
```

1.1.10 Absolute value

The absolute value function applied to a number `x` can be written either `abs(x)` or using the traditional notation `|x|`.

```
Disco> abs(6)
6
Disco> abs(-6)
6
Disco> abs(-1/2)
1/2
Disco> |-6|
6
```

Since the output of `abs` is always positive, it can convert *rational numbers* into *fractional numbers*, and *integers* into *natural numbers*.

The notation `|~|` can also be used to find the *size* of a collection such as a *set*, bag, or list.

1.1.11 Combinatorics

Disco has a growing collection of operations relating to *combinatorics*, *i.e.* counting things.

Factorial

Factorial, written `!`, is a *unary operator* written after its argument, defined as the product of all the natural numbers from 1 up to n, that is, $n! = 1 \times 2 \times 3 \times \cdots \times n$.

```
Disco> :doc !
~! : ℕ → ℕ

n! computes the factorial of n, that is, 1 * 2 * ... * n.

https://disco-lang.readthedocs.io/en/latest/reference/factorial.html

Disco> 3!
6
Disco> 4!
24
Disco> 4! == 1 * 2 * 3 * 4
true
Disco> (4!)!
620448401733239439360000
Disco> ((4!)!)!
Error: that number would not even fit in the universe!
```

(continues on next page)

(continued from previous page)

```
Disco> 0!
1
```

Note that $0! = 1$ by definition, since a product of zero things should be the identity value for multiplication.

Binomial and multinomial coefficients

The *binomial coefficient* $\binom{n}{k}$ represents the number of different ways to choose a subset of size k out of a set of size n, and is in general given by the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

However, binomial coefficients can be computed more efficiently than literally using the above formula with *factorial*, so Disco has special built-in support for computing them. Since $\binom{n}{k}$ is usually pronounced "n choose k", the Disco syntax is `n choose k`. For example:

```
Disco> 5 choose 2
10
Disco> 7 choose 0
1
Disco> 0 choose 0
1
Disco> 7 choose 8
0
Disco> 100 choose 23
24865270306254660391200
```

Multinomial coefficients

Disco also has support for *multinomial coefficients*:

$$\binom{n}{k_1 \quad k_2 \quad \ldots \quad k_r} = \frac{n!}{k_1!k_2!\ldots k_r!(n-k_1-k_2-\cdots-k_r)!}$$

is the number of ways to simultaneously choose subsets of size $k_1, k_2, \ldots, k_r$ out of a set of size n. In Disco, a multinomial coefficient results when the second argument to `choose` is a list instead of a natural number. For example:

```
Disco> 10 choose 2
45
Disco> 10 choose [2]
45
Disco> 10 choose [2,3]
2520
Disco> 10 choose [2,3,5]
2520
Disco> 10 choose [2,3,5] == (10 choose 2) * (8 choose 3) * (5 choose 5)
true
```

1.2 Comparison

Comparison operators can be used to compare two values, to test if they are equal or to see what order they are in. Comparison operators always return a *boolean* value.

Values of almost any type can be compared: *numeric types*, *booleans*, *characters*, strings, and any *pair*, *sum*, or container types built out of these. For example, sets of pairs of natural numbers and strings can be compared:

```
Disco> {(3, "hi"), (4, "there"), (6, "world")} < {(10, "what")}
true
```

See the individual pages about each type for more information on how comparison works on values of that type.

Functions, on the other hand, cannot be compared, because in general this would require testing the functions on every single possible input, of which there might be infinitely many.

```
Disco> (\n:N. n) < (\n:N. n+1)
Error: values of type a2 → a3 cannot be compared.
https://disco-lang.readthedocs.io/en/latest/reference/not-qual.html
```

- The == operator is for testing equality. For example,

  ```
  Disco> 3 == 5
  false
  Disco> "hi" == "hi"
  true
  ```

  Equality is one critical point where Disco syntax has to deviate from standard mathematical notation: be sure to keep in mind the difference between = (which is used to *define things*) and ==, used for testing whether two things are equal. For more information on the difference, see the page on *definition vs equality testing*.

- The /= operator (also written or !=) means "not equal to". It is true if and only if its two arguments are not equal. The syntax /= is supposed to remind us of the standard mathematical notation of an equality sign with a slash through it (). However, != is also provided for those used to this operator in other programming languages.

- There are four operators that can be used to test the ordering of two values:
  - < tests whether the first value is less than the second.
  - > tests whether the first value is greater than the second.
  - <=, also written or =<, tests whether the first value is less than or equal to the second.
  - >=, also written or =>, tests whether the first value is greater than or equal to the second.

- You can chain multiple comparisons; this always means the same thing as combining all the individual comparisons with "and". For example, `3 <= x < y <= 10` means the same thing as `3 <= x /\ x < y /\ y <= 10`.

1.3 Logical operations

Disco has various standard operations for manipulating *Boolean values*.

- Logical negation is written `not`; it inverts `true` to `false` and vice versa.
- Logical conjunction, aka AND, is written `/\`, `and`, or `&&`. It has the following truth table:

x	y	x /\ y
F	F	F
F	T	F
T	F	F
T	T	T

- Logical disjunction, aka OR, is written `\/`, `or`, or `||`. It has the following truth table:

x	y	x \/ y
F	F	F
F	T	T
T	F	T
T	T	T

- Logical implication, aka IF-THEN, is written `->`, `==>`, or `implies`. It has the following truth table:

x	y	x -> y
F	F	T
F	T	T
T	F	F
T	T	T

- Biconditional, aka "if and only if", is written `<->`, `<==>`, or `iff`. It has the following truth table:

x	y	x <-> y
F	F	T
F	T	F
T	F	F
T	T	T

1.4 Syntax

Syntax refers to the rules for writing a language. For example, the syntax of the English language includes things like spelling and grammar. The syntax of Disco programs likewise refers to "spelling and grammar", that is, the various ways to write valid Disco programs.

1.4.1 Variables

A *variable* is a name given to some value. Variable names can contain lowercase and uppercase letters, digits, underscores (_) and apostrophes, with the only restriction that a name must start with a letter. For example, `myHorse3`, `some_name`, and `X__17'x'_` are all valid variable names.

To define a variable, one must first use a *type signature* to declare its type on a line by itself, like

```
variable_name : type
```

where `type` is replaced by whatever type the variable should have. The value of the variable can then be *defined* using an = sign, like

```
variable_name = expr
```

where `expr` represents an arbitrary *expression*. (There is also special syntax available for defining *functions*.) For example:

```
my_variable : Z
my_variable = 2 * 7 + 9
```

The above code defines the variable `my_variable` with the type `Z` (*i.e.* an *integer*) and the value 23.

1.4.2 Type signature

A *type signature* declares what *type* a *variable* has. It consists of a variable name, a colon, and a type on one line together. For example,

```
x : Z
```

is a type signature declaring the variable `x` to have type `Z`.

Every *definition* must have a type signature that comes before it.

1.4.3 Expressions

An *expression* is some combination of values, operators, and functions which describes a *value* (turning an expression into a value is called *evaluation*).

Examples of expressions in Disco include:

- `5` (a single *number*, string, *boolean*, *variable*, etc. by itself is an expression)
- `1 + 2` (two or more expressions combined by *operators* is again an expression)
- `f(x,3) * g(2)` (*function* calls are expressions)

Examples of things which are *not* expressions include:

- `x : N` (this is a *type signature*, not an expression; it does not have a value, it says what the type of `x` is)
- `x = 3` (this is a *definition*)

Warning: Be careful not to confuse `x = 3` (a *definition* of the *variable* `x`) with `x == 3` (a *comparison* expression which has a value of either `true` or `false` depending on whether `x` is equal to `3` or not). See *Definition versus equality testing*.

1.4.4 Definitions

We can define a *variable* to have a certain value using the syntax

```
variable = expression
```

Note that every definition also must have a *type signature* before it. So, for example,

```
x : Z
x = -17
```

declares the variable `x` to have the type `Z` and to represent the value `-17`. From now on, whenever we use `x`, it can be thought of as an abbreviation for the number `-17`.

Note that the equals sign in Disco really means *mathematical equality*, like an equation in algebra, and that a variable *can have only one definition*. If you are already familiar with an imperative language like Python or Java, read the next section for a comparison with Disco. If Disco is your first programming language, you can skip this (though you may read it if you are interested).

Definition vs assignment

In many *imperative* languages, variables can be thought of as "boxes" that store values, and the equals sign means *assignment*. For example, in Python,

```
x = 5
x = 7
```

means that we should first assign the value `5` to the variable `x`; then, we *replace* the value stored by `x` with `7`.

In contrast, in Disco (as in some other *functional* languages), variables are *names* for values, and the equals sign means *definition*. In Disco,

```
x = 5
x = 7
```

is an error, because `x` cannot be defined as both `5` and `7`; it cannot be equal to both at the same time. In other words, it is like a system of two equations with no solution.

1.4.5 Definition versus equality testing

In standard mathematical notation, the = symbol can be used in at least two distinct (yet related) contexts:

- To *define* things, as in, "Let $x = 3y + 2$, and consider…" In this example sentence, $x = 3y + 2$ *defines* the variable `x` as standing for the expression `3y + 2`.
- As a *relation* which can hold, or not, as in, "If $x = 3y + 2$, then … but otherwise …". In this example sentence, x and y must already be defined, and $x = 3y + 2$ is something that is either true or false.

Notice how the exact same expression $x = 3y + 2$ is used in both examples, but means two very different things depending on the context—which is defined entirely by the English words surrounding the symbols! Disco does not have the luxury of using English words to figure out what we mean; instead, Disco must use two different symbols. The = symbol is used to express *definitions*, as in the first example; whereas the == symbol *tests whether two things are equal*, as in the second example.

1.4.6 Operators

An *operator* is a function that is written in a special way. Normal functions are written before their arguments, like `f(x,y)`. *Binary* (two-argument) operators are symbols or words which are written *in between* their two arguments, like `1 + 2`. Disco has many built-in operators which are symbols (like `+`, `*`, `/`, *etc.*) as well as a few which are words (like `mod`, `choose`, and `divides`).

Disco also has three *unary* operators: arithmetic negation (`-`) and logical negation (`¬` or `not`) are written in front of their one argument, and *factorial* (`!`) is written after its argument.

When multiple operators are used together, Disco uses their *precedence and associativity* to decide how to interpret it.

Twiddle notation

Disco has a special syntax for talking about operators on their own, without any arguments: a tilde (or "twiddle") (~) goes in each place where an argument would be. For example, to talk about the + operator on its own we can write ~+~. To talk about the *factorial* operator we would write ~!, because factorial only takes a single argument which goes before it. Disco will use this "twiddle notation" when you ask it for the type of an operator:

```
Disco> :type !
~! :  →
Disco> :type ~!
~! :  →
```

Note that in this case, we can write ! or ~! and Disco understands either one.

The twiddle notation is also useful when giving an operator as an argument to a higher-order function:

```
Disco> reduce(~+~, 0, [1 .. 10])
55
```

Operator documentation

You can ask for *documentation* about operators directly, for example:

```
Disco> :doc !
~! :  →

n! computes the factorial of n, that is, 1 * 2 * ... * n.

https://disco-lang.readthedocs.io/en/latest/reference/factorial.html
```

1.4.7 Operator precedence and associativity

When we write something like $1 + 2 \times 3$, how do we know what it means? Does it mean $(1 + 2) \times 3$, or $1 + (2 \times 3)$? Of course, you are familiar with the usual "order of operations", where multiplication comes before addition, so in fact $1 + 2 \times 3$ should be interpreted as $1 + (2 \times 3) = 1 + 6 = 7$.

Another way to say this is that multiplication has *higher precedence* than addition. If we think of operators as "magnets" that attract operands, higher precedence operators are like "stronger magnets".

Another issue arises when operators are repeated, or when operators with the same precedence are used together. For example, does $4 - 3 - 2 - 1$ mean $((4 - 3) - 2) - 1$ or $4 - (3 - (2 - 1))$? In fact, it means the former, because addition and subtraction are done "left to right"; we say they are *left associative*. On the other hand, exponentiation is *right associative*, meaning that `1 ^ 2 ^ 3 ^ 4 = 1 ^ (2 ^ (3 ^ 4))`.

Every operator in Disco has a precedence level and associtivity, and Disco uses these to determine where to put parentheses in expressions like `1 + 2 * 3` or `5 > 2 ^ 2 + 1 and 7 > 2 ==> true`. You might have memorized something like PEMDAS, but Disco has so many operators that memorizing their precedence levels is out of the question! Instead, we can use the `:doc` command to show us the precedence level and associativity of different operators.

```
Disco> :doc ^
~^~ :  ×  →
precedence level 13, right associative

Disco> :doc +
```

(continues on next page)

(continued from previous page)

```
~+~ :  ×  →
precedence level 7, left associative

Disco> :doc >
~>~ :  ×  → Bool
precedence level 5, right associative

Disco> :doc and
~and~ : Bool × Bool → Bool
precedence level 4, right associative

Disco> :doc ==>
~==>~ : Bool × Bool → Bool
precedence level 2, right associative
```

1.4.8 Case expressions

We can use "case expressions" to choose among multiple alternatives.

Basic case expressions with conditions

The basic form of a case expression is as follows:

```
{? alternative1   if  condition1,
   alternative2   if  condition2,
   ...
   alternativeN   otherwise
?}
```

This will try each condition, starting with the first, until finding the first condition that is true. Then the value of the entire case expression will be equal to the corresponding alternative. The `otherwise` case will always be chosen if it is reached. Each condition must be an expression of type *Bool*; the alternatives can have any type (though they must all have the *same* type, whatever it is).

For example, consider the definition of the `caseExample` function below:

```
caseExample : N -> N
caseExample(n) =
  {? n + 2    if n < 10 \/ n > 20,
     0        if n == 13,
     77n^3    if n == 23,
     n^2      otherwise
  ?}
```

Here are a few sample inputs and outputs for `caseExample`, with an explanation of each:

- `caseExample(5) == 7`: the first condition is true (since `5 < 10`), so the result is `5 + 2`.
- `caseExample(23) == 25`: the first condition is again true (since `23 > 20`), so the result is `23 + 2`. Note that the first true condition is always chosen, so it does not matter that the later condition `n == 23` would also be true.
- `caseExample(13) == 0`: the first condition is false (13 is neither `< 10` nor `> 20`, but the second condition (`13 == 13`) is true.

- `caseExample(12) == 144`: the first three conditions are all false, so the `otherwise` case is used, with the result `12^2`.

If *none* of the conditions in a case expression are true, it is an error: see *Value did not match any of the branches in a case expression*.

Case expressions with conditions and patterns

More generally, case expressions can use *pattern matching* in addition to Boolean conditions, and each alternative in a case expression can have multiple conditions. The most general form of a case expression is as follows:

```
{? alternative1   guard11 guard12 ...,
   alternative2   guard21 guard22 ...,
   ...
?}
```

where each `guard` has one of two forms:

- `if <condition>`. This guard succeeds if the condition is true.
- `if <expression> is <pattern>`. This guard succeeds if the given *expression* matches the *pattern*; furthermore, any *variables* in the pattern will be defined locally within the corresponding `alternative` as well as any subsequent guards in the same clause.

The keyword `when` can also be used as a synonym for `if`.

1.4.9 Comments

Comments in Disco can be written using `--`. Anything from `--` to the end of the line is a *comment* which Disco will ignore. Comments can be used to write notes to yourself or explanations to others, to write your name in a `.disco` file, and so on.

```
Disco> 1 + 2 -- Disco will ignore this
3
```

See also *documentation*, which is a special kind of comment using `|||` instead of `--`.

1.4.10 Documentation

Disco allows special *documentation comments*, which begin with three vertical bars (`|||`) at the beginning of a line. Anything written after `|||` is documentation which will be attached to the next definition (either a type definition or a *variable* definition). This documentation can later be accessed with the `:doc` command. For example, suppose we have the following in a file called `cool.disco`:

Listing 1: example/cool.disco

```
-- This is a comment that will be ignored.
||| f is a cool function which computes a thing.
||| It has two lines of documentation.
f : N -> N
f(x) = x + 7
```

Then at the disco prompt we can load the file, and see the documentation for `f` using the `:doc` command:

```
Disco> :load cool.disco
Loading cool.disco...
Loaded.
Disco> :doc f
f :  →

f is a cool function which computes a thing.
It has two lines of documentation.
```

1.5 Types

Every *expression* in disco has a *type*, which tells us *what kind of value* will result when we evaluate the expression. For example, if an expression has type `N`, it means we are guaranteed to get a *natural number* as a result once the expression is done being evaluated.

The type of an expression thus represents a *promise* or *guarantee* about the behavior of a program. Checking that all the types in a program match up can also be seen as a way of *predicting* or *analyzing* the behavior of a program without actually running it.

Each type can be thought of as a collection of values which all have a similar "shape".

The type of each variable in Disco must be declared with a *type signature*. We can also give Disco hints about the intended type of an expression using a type annotation. We can define our own new types using a type definition.

In some situations, Disco may be willing to accept something of one type when it was expecting another: specifically, when the given type is a *subtype* of the one it was expecting.

1.5.1 Base types

Base types are the fundamental types that define the possible kinds of simple data values in Disco.

Booleans

The type of *booleans* is written `Bool` or `Boolean`. There are exactly two values of type `Boolean`: `false` and `true` (which can also be written `False` and `True`).

Logical operators can be used to manipulate `Boolean` values, and expressions of type `Boolean` can be used as conditions in a *case expression*.

Numeric types

Natural numbers

The type of *natural numbers* is written `N`, , `Nat`, or `Natural` (Disco always prints it as , but you can use any of these names when writing code). The natural numbers include the counting numbers 0, 1, 2, 3, 4, 5, . . .

Adding or *multiplying* two natural numbers yields another natural number:

```
Disco> :type 2 + 3
5 :
Disco> :type 2 * 3
6 :
```

Natural numbers cannot be directly *subtracted* or *divided*. However, `N` is a subtype of all the other numeric types, so using subtraction or division with natural numbers will cause them to be automatically converted into a different type like *integers* or *rationals*:

```
Disco> :type 2 - 3
2 - 3 :
Disco> :type 2 / 3
2 / 3 :
```

Note that some mathematicians use the phrase "natural numbers" to mean the set of positive numbers 1, 2, 3, . . . , that is, they do not include zero. However, in the context of computer science, "natural numbers" almost always includes zero.

Integers

The type of *integers* is written `Z`, , `Int`, or `Integer`. The integers include the positive and negative counting numbers (as well as zero): . . . , -3, -2, -1, 0, 1, 2, 3, . . .

Adding, *multiplying*, or *subtracting* two integers yields another integer. Trying to *divide* two integers will automatically convert the result to a *rational number*:

```
Disco> :type 2 * (-3)
2 * (-3) :
Disco> :type 2 / (-3)
2 / (-3) :
```

Fractional numbers

The type of *fractional numbers* is written `F`, , `Frac`, or `Fractional`. The fractional numbers include all the natural numbers (0, 1, 2, . . .) along with all the positive fractions formed from the ratio of two natural numbers (such as 1/2, 13/7, 56/57, . . .)

Adding, *multiplying*, or *dividing* two fractional numbers yields another fractional number. Trying to *subtract* two fractional numbers will automatically convert the result to a *rational number*:

```
Disco> :type (1/2) * (2/3)
1 / 2 * 2 / 3 :
Disco> :type 1/2 - 2/3
1 / 2 - 2 / 3 :
```

The special sets (natural numbers), (integers), and (rational numbers) are very common in mathematics and computer science, but the set of fractional numbers is not common at all (in fact, I made up the name and the notation). People usually start with the natural numbers , extend them with subtraction to get the integers , and then extend those again with division to get the rational numbers . However, there is no reason at all that we can't do it in the other order: first extend the natural numbers with division to get the fractional numbers , then extend with subtraction to get . Having all four types in Disco (even though one of them is not very common in mathematical practice) makes many things simpler and more elegant.

Rational numbers

The type of *rational numbers* is written `Q`, , or `Rational`. The rational numbers can be thought of as the combination of and , and include zero, positive and negative counting numbers, and positive and negative fractions.

```
Disco> :type -3/5
-3 / 5 :
```

Doing any of the four *arithmetic* operations (*addition*, *multiplication*, *subtraction* or *division*) on two rational numbers results in another rational number.

Note that Disco does not have any way to work with *irrational* numbers such as π or $\sqrt{2}$; such numbers are typically not needed in discrete mathematics, and including them would make the language much more complicated.

Disco has four *types* which represent numbers:

- *Natural numbers*, written `N`, , `Nat`, or `Natural`. These represent the counting numbers 0, 1, 2, ... which can be added and multiplied.

  ```
  Disco> :type 5
  5 :
  ```

- *Integers*, written `Z`, , `Int`, or `Integer`, allow negative numbers such as `-5`. They extend the natural numbers with subtraction.

  ```
  Disco> :type -5
  -5 :
  ```

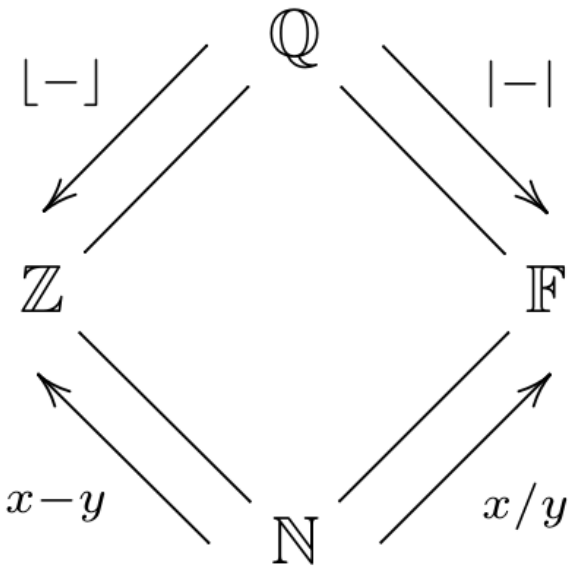
- *Fractional numbers*, written `F`, , `Frac`, or `Fractional`, allow fractions like `2/3`. They extend the natural numbers with division.

  ```
  Disco> :type 2/3
  2 / 3 :
  ```

- *Rational numbers*, written `Q`, , or `Rational`, allow both negative and fractional numbers, such as `-2/3`.

  ```
  Disco> :type -2/3
  -2 / 3 :
  ```

We can arrange the four numeric types in a diamond shape, like this:



Each type is a subset, or subtype, of the type or types above it. For example, the fact that $\mathbb{N}$ is below $\mathbb{Z}$ means that every natural number is also an integer.

- The values of every numeric type can be *added* and *multiplied*.
- The arrow labelled $x - y$ indicates that going up and to the left in the diamond (*i.e.* from $\mathbb{N}$ to Z or F to Q) corresponds to adding the ability to do subtraction. That is, values of types on the upper left of the diamond ($\mathbb{Z}$ and $\mathbb{Q}$) can also be *subtracted*.
- Going up and to the right corresponds to adding the ability to do division; that is, values of the types on the upper right of the diamond ($\mathbb{F}$ and $\mathbb{Q}$) can also be *divided*.
- To move down and to the right (*i.e.* from $\mathbb{Z}$ to $\mathbb{N}$, or from $\mathbb{Q}$ to $\mathbb{F}$), you can use *absolute value*.
- To move down and to the left (*i.e.* from $\mathbb{F}$ to $\mathbb{N}$, or from $\mathbb{Q}$ to $\mathbb{Z}$), you can take the *floor or ceiling*.

Characters

The type of *characters* is written `Char`. Values of this type represent single characters (*e.g.* a letter, a digit, punctuation, etc.). Technically, a `Char` value represents any Unicode *codepoint*.

Characters are written using single quotes. For example,

```
mychar : Char
mychar = 'g'
```

Certain special characters can be written using *escape sequences*, consisting of a backslash followed by another character:

- `\n` represents a newline character
- `\t` represents a tab character
- `\'` represents a single quote
- `\"` represents a double quote
- `\\` represents an actual (single) backslash character

A sequence of characters is known as a string.

1.5.2 Function types

Function types represent *functions* that take inputs and yield outputs. The *type* of functions that take inputs of type *A* and yield outputs of type *B* is written `A -> B` (or `A → B`).

Every function in Disco takes exactly one input and produces exactly one output. "Multi-argument" functions can be represented using a *pair type* as the input type. For example, the type of a function taking two natural numbers as input and producing a rational number as output could be written `N × N → Q`.

Note that if `A` and `B` are types, then `A -> B` is itself a type, which can be used in all the same ways as any other type. In other words, functions in Disco are "first-class", which means they can be used in the same ways as any other values: for example, functions *can be given as input to other functions*, or returned as output, we can have lists of functions, *etc.*

Function values cannot be *compared*, however, because in general this would require testing the functions on every single possible input, of which there might be infinitely many.

1.5.3 Polymorphism

Sometimes, we want to express the fact that a certain function will work for *any* input type at all. For example, consider a function to find the length of a list of natural numbers:

```
lengthN : List(N) -> N
lengthN([]) = 0
lengthN(_ :: xs) = 1 + lengthN(xs)
```

Or how about a function to find the length of a list of rational numbers:

```
lengthQ : List(Q) -> Q
lengthQ([]) = 0
lengthQ(_ :: xs) = 1 + lengthQ(xs)
```

It is easy to see that `lengthN` and `lengthQ` are identical except for their types, and that it is going to be very tedious if we have to write a different version of this function for every possible element type. The length of a list does not depend on the elements at all, so we would like to be able to define it once and for all, in a way that will work for any type of list.

Indeed, we can do exactly that, by using a *type variable* (any name that starts with a lowercase letter) in place of the concrete element type:

```
length : List(a) -> N
length([]) = 0
length(_ :: xs) = 1 + length(xs)
```

Here, the variable `a` can stand for any type. This expresses both a *requirement* that the definition of `length` does not care what type `a` is (and disco will actually check to make sure that is the case), and a *promise* that `length` can be used on any particular list. For example,

```
Disco> length [1,2,3]
3
Disco> length [True, False]
2
```

On the other hand, suppose we tried to define this function, which adds one to every element of a list:

```
incr : List(a) -> List(a)
incr([]) = []
incr(x :: xs) = (x + 1) :: incr(xs)
```

Disco does not accept this definition. The problem is that although it promises that it will work on lists of any type, it doesn't: it tries to add one to the elements, and adding 1 is only something that works for some types. For example, it doesn't make sense to add 1 to every element in a list of Booleans. `incr` will work fine if we give it a more specific type, such as `List(N) -> List(N)`.

Another good example is *function composition*, which takes two functions and connects the output of one function to the input of the other, creating a new function representing the "pipeline" of doing one function then the other. The input and output types of the functions don't matter at all — other than the fact that the output type of the one function has to match the input type of the other. We can write it as follows:

```
compose : (b -> c) * (a -> b) -> (a -> c)
compose(f,g) = \x. f(g(x))
```

`a`, `b`, and `c` can all stand for different types (although they are not *required* to be different). Notice, however, that the input type of the first function is `b`, and the output type of the second function is also `b`—hence no matter what type `b`

represents, they must be the same. The function that results takes an input of type `a` and ultimately produces an output of type `c` after running the input through both functions.

Note that type definitions can also be polymorphic; see that page for more information.

1.5.4 Algebraic types

Algebraic types are the building blocks that let us build up more complex types.

Unit type

`Unit` is a special built-in type with only a single value, which can be written `unit` (or , U+25A0 BLACK SQUARE).

```
Disco> :type unit
 : Unit
Disco> unit

Disco>
```

This is not very useful on its own, but becomes very useful when combined with *sum* and *pair types* to create custom recursive *algebraic types*.

Pair types

Pair types, or *product types*, represent *ordered pairs* of values. Suppose `A` and `B` are *types*. Then:

- `A * B` (also written `A × B`) is a *pair type* (also known as a *product type* or *Cartesian product*). It represents the set of all possible pairs where the first element has type `A` and the second element has type `B`.
- A pair is written `(a, b)` (where `a` and `b` can be arbitrary *expressions*). Specifically, if `a : A` and `b : B`, then the ordered pair `(a, b)` has type `A * B`. For example:

  ```
  Disco> :type (1, True)
  (1, true) :  × Bool
  Disco> :type (-7, -3)
  (-7, -3) :  ×
  ```

Pair types are commonly used to represent functions that take multiple inputs. For example, `f : N * Z -> Q` means that `f` takes a *pair* of a natural number and an integer as input. Such functions are often defined via *pattern matching* on the pair, *like so*:

```
f : N * Z -> Z
f(n,z) = 3n - z
```

n-tuples and nested pairs

We have seen that `A * B` is a type of *pairs* of values. What about triples, quadruples, ... n-tuples of values? The answer is simple:

- triples are written `(x,y,z)` and have types like `A * B * C`;
- quadruples are written `(w,x,y,z)` and have types like `A * B * C * D`;
- and so on.

So, for example, a function taking a quintuple of values could be written like this:

```
funTaking5Tuple : N * Z * List(N) * Q * Bool -> Int
funTaking5Tuple(n,z,l,q,b) = ...
```

Note: General n-tuples actually are not specially built in at all: rather, everything is actually built out of *nested pairs*. For convenience, pair types and values both *associate to the right*, that is,

- the type `A * B * C` is interpreted as `A * (B * C)`, and
- correspondingly, `(x, y, z)` is interpreted as `(x, (y, z))`.

So, for example, the definition of the function `funTaking5Tuple` from above is really shorthand for

```
funTaking5Tuple : N * (Z * (List(N) * (Q * Bool))) -> Int
funTaking5Tuple(n,(z,(l,(q,b)))) = ...
```

Typically one can just use triples or 5-tuples or whatever and not think about this, but occasionally it's helpful to understand the way things are represented with nested pairs under the hood.

Sum types

Sum types represent situations where we have a value which could be *either one thing or another*. Suppose `A` and `B` are *types*. Then:

- `A + B` is a *sum type* (also known as a *disjoint union*). It represents a *disjoint union* of the types `A` and `B`. That is, the values of `A + B` can be either a value of type `A`, or a value of type `B`.
- A value of type `A + B` can be written either `left(a)`, where `a` is an arbitrary *expression* of type `A`, or `right(b)`, where `b` is an arbitrary expression of type `B`. For example:

```
Disco> left(3) : N + Bool
left(3)
Disco> right(false) : N + Bool
right(false)
```

Note that the `left` or `right` ensures that `A + B` really does represent a *disjoint* union. For example, although the usual union operator is idempotent, that is, $\mathbb{N} \cup \mathbb{N} = \mathbb{N}$, with a disjoint union of types `N + N` is not at all the same as `N`. Elements of `N + N` look like either `left(3)` or `right(3)`, that is, `N + N` includes *two* copies of each natural number.

1.5.5 Collection types

Collection types represent various ways to store *collections* of values.

Sets

For any *type* `T`, `Set(T)` is the type of *finite sets* with elements of type `T`.

- The empty set is written `{}`.
- A set with specific elements can be written like this: `{1, 2, 3}`.
- An *ellipsis* can be used to generate a range of elements. For example,

```
Disco> {1 .. 5}
{1, 2, 3, 4, 5}
Disco> {1, 3 .. 9}
{1, 3, 5, 7, 9}
```

- *Set comprehension* notation can also be used, for example:

```
Disco> {x^2 + 1 | x in {1 .. 10}, x > 4}
{26, 37, 50, 65, 82, 101}
```

- The built-in `set` function can be used to convert other collections (*e.g.* lists) to sets:

```
Disco> set([1,2,3,2,3])
{1, 2, 3}
Disco> set("hello")
{'e', 'h', 'l', 'o'}
```

The order of elements in a set does not matter, nor does the number of copies of an element. For example,

```
Disco> {3,3,1,2} == {1,1,2,2,3,3}
true
Disco> {3, 3, 1, 2}
{1, 2, 3}
```

To check whether a set contains a given element, one can use the `elem` operator (also written):

```
Disco> 2 elem {1,2,3}
true
Disco> 5 elem {1,2,3}
false
Disco> 2  {1,2,3}
true
```

Sets support various operations, including *size*, *union*, *intersection*, *difference*, *subset*, and *power set*.

1.5.6 Propositions

`Prop` is the type of *propositions*. A proposition is a statement that could be true or false. `Prop` is very similar to `Bool`; the main difference is that whereas any `Bool` can always be evaluated to see if it is `true` or `false`, this may not be possible for a `Prop` due to the use of quantifiers (`forall` and `exists`).

Any *Boolean* expression can be used as a proposition. For example, `true`, `x > 5`, and `(x < y) -> ((x == 2) \/ (x == 3))` can all be used as propositions.

However, unlike Boolean expressions, propositions can use quantifiers, that is, `forall` or `exists`. For example, `forall x :  Z. x^2 >= 0` is a `Prop` (but not a `Bool`).

If you type a proposition at the Disco prompt, it will simply print `<Prop>` and refuse to evaluate it. If you want to get an idea of whether a proposition is true or false, you have two options:

- The built-in `holds` function tries its best to determine whether a proposition is true or false. If it returns a value at all, it is definitely correct. For example,
  - `holds ((5 < 3) \/ ((2 < 1) -> false))` yields `true`
  - `holds (forall p:Bool. forall q:Bool. (p \/ q) <-> (q \/ p))` yields `true`
  - `holds (forall p:Bool. forall q:Bool. (p \/ q) <-> (p /\ q))` yields `false`

  - `holds (forall n:N. n < 529)` yields `false`

  However, sometimes it may simply get stuck in an infinite loop. For example, `holds (forall n:N. n >= 0)` will simply get stuck. Even though it is obvious to us that this proposition is true, `holds` is not smart enough to see this; it simply tries evaluating `n >= 0` for every natural number, which will never finish.

- One can also use the `:test` command. This command makes a best effort to evaluate a proposition, but only trying a finite number of examples for infinite domains. Additionally, in many cases it can output much more information than a simple `true` or `false`, for example, showing a counterexample if a `forall` is false, or a witness if an `exists` is true. For example,
  - `:test forall p:Bool. forall q:Bool. (p \/ q) <-> (p /\ q)` prints

```
- Certainly false: p. q. p \/ q <-> p /\ q
  Counterexample:
    p = false
    q = true
```

  - However, `:test forall n:N. n < 529` prints

```
- Possibly true: n. n < 529
  Checked 100 possibilities without finding a counterexample.
```

    Obviously this proposition is false, but Disco apparently does not try a big enough value of `n` to be able to tell.

Note that at the moment it is not possible to combine propositions using *logical operators* like `/\`, `\/`, or `->`. For example, Disco does not currently accept something like `(forall x : N. x >= 0) /\ (exists x : N. x == 3)`, although this is certainly a proposition from a mathematical point of view. This is something that may be added in the future.

1.5.7 Subtypes

In some cases, Disco is willing to accept one type in place of another, when this is known to be safe. For example, suppose we have the following definitions:

```
f : N -> N
f(n) = sqrt(n) + 1

g : Z -> Z
g(n) = n - 5

x : N
x = 16

y : Z
y = -10
```

Of course we can give `x` as an input to `f`, because the type of `f` says it takes natural numbers as input, and `x` is a natural number. Likewise, the type of `y` matches `g`'s input type, so we can give `y` as an input to `g`:

```
Disco> f(x)
5
Disco> g(y)
-15
```

On the other hand, we cannot give `y` as an input to `f`. `f` is expecting only natural numbers as input, and it might not be safe to give it a negative number. In fact, in this case, it's definitely not safe: we cannot take the square root of a negative number.

```
Disco> f(y)
Error: typechecking failed.
https://disco-lang.readthedocs.io/en/latest/reference/typecheck-fail.html
```

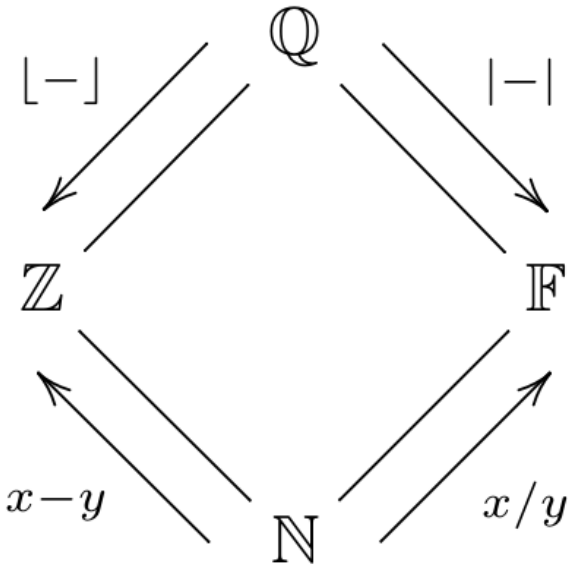
However, we *can* give `x` as an input to `g`:

```
Disco> g(x)
11
```

Why is that? Well, mathematically speaking, every natural number is also an integer, so if a function is prepared to receive any integer (positive or negative) as input, then giving it only natural numbers (*i.e.* nonnegative integers) is perfectly safe. Intuitively, disco automatically "converts" the natural number into an integer before giving it to `g`.

Since it is always safe to use a natural number anywhere an integer is expected, we say that `N` is a *subtype* of `Z`.

The four basic *numeric types* can be arranged in a diamond shape, like so:



Each type is a subtype of the type or types above it. That is, `Z` is a subtype of `Q`, `F` is a subtype of `Q`, and `N` is a subtype of all the others.

1.5.8 Subtyping for algebraic types

In addition to the subtype relationships between the basic numeric types, more complex algebraic types can be subtypes of each other too.

- For *pair types*, `A * B` is a subtype of `C * D` exactly when `A` is a subtype of `C`, and `B` is a subtype of `D`. For example, given these definitions:

```
g : Z * Q -> Q
g(x,y) = y / (3x)
```

(continues on next page)

(continued from previous page)

```
p : N * N
p = (2,5)
```

It is allowed to give `p` as an input to `g`:

```
Disco> p(g)
5/6
```

Since `N` is a subtype of `Z`, and `N` is a subtype of `Q`, therefore `N * N` is a subtype of `Z * Q`.

- Similarly, for *sum types*, `A + B` is a subtype of `C + D` exactly when `A` is a subtype of `C`, and `B` is a subtype of `D`. For example, given these definitions:

```
g : Z + Q -> Q
g(left(z)) = z / 2
g(right(y)) = 3y

p : N + Z
p = left(1)

r : N + Z
r = right(-2)
```

It is allowed to give `p` and `r` as inputs to `g`:

```
Disco> g(p)
1/2
Disco> g(r)
-6
```

Since `N` is a subtype of `Z`, and `N` is a subtype of `Q`, therefore `N + N` is a subtype of `Z + Q`.

- Function types work a little differently. `A -> B` is a subtype of `C -> D` exactly when `C` **is a subtype of** `A` and `B` is a subtype of `D`. Notice how the relationship is reversed for the input types. Working out why this makes sense is left as an interesting exercise for the reader.

1.6 Functions

1.6.1 Pattern matching

Functions in Disco can be defined using *pattern matching*, which in general looks like this:

```
f(pattern) = expression
```

This means, roughly, "if the input to the function `f` looks like `pattern`, then the output of `f` on that input should be `expression`." For example, `f(5) = 29` means that if the input to `f` is the number `5`, then the output should be `29`.

Functions can be defined by multiple pattern-match *clauses*; Disco tries the clauses in order, one by one, and picks the first one that matches. (Note that *case expressions* can also be used to define functions, in case more sophisticated logic is needed.) For example,

```
f(2) = 12
f(2k) = k+1
f(n) = 2n+1
```

means:

- First, if the input to `f` is specifically `2`, then return `12`;
- next, if the input to `f` is even (*i.e.* of the form `2k` for some integer `k`), return `k+1`;
- finally, for any other input, which we will call `n`, return `2n+1`.

(If none of the clauses in a function definition matches an input, it is an error: see *Value did not match any of the branches in a case expression*.)

The above example uses a *literal pattern*, an *arithmetic pattern*, and a *variable pattern*; see the links below for more specific information about the different types of patterns that can be used.

Literal pattern

A literal *pattern* consists of a single specific value. For example,

```
f(5) = "hi"
```

uses `5` as a literal pattern, and means that on the specific input `5`, the function `f` should output `"hi"`.

Literal patterns can be used with

- The *unit value*, *e.g.* `f(unit) = ...`
- *Booleans*, *e.g.* `f(true) = ...`
- *Natural numbers*, *e.g.* `f(5) = ...`
- *Integers*, *e.g.* `f(-5) = ...`
- *Rational numbers*, *e.g.* `f(1/2) = ...`
- *Characters*, *e.g.* `f('x') = ...`
- Strings, *e.g.* `f("hello") = ...`

Note that `f(-5) = ...` and `f(1/2) = ...` are technically *arithmetic patterns* rather than true literal patterns, but the distinction does not matter very much.

Variable pattern

A variable *pattern* simply consists of a single variable. It always successfully matches any input; within the corresponding clause, the input can be referred to by the variable name. For example,

```
f(n) = 3n + 1
```

means "for *any* input to the function `f`, which we will call `n`, output the value which is one more than three times the input `n`."

The variable defined by a variable pattern is *local* to the clause and cannot be referenced anywhere else. For example:

```
Disco> f : N -> N
Disco> f(n) = 3n+1
Disco> f(3)
10
Disco> n
Error: there is nothing named n.
https://disco-lang.readthedocs.io/en/latest/reference/unbound.html
```

This also means multiple function definitions can use the same variable name without interfering with one another at all.

Wildcard patterns

A wildcard *pattern* is just an underscore character, indicating that we do not care about a particular input. Just like a *variable pattern*, it successfully matches any input; unlike a variable pattern, it does not define a new name.

For example,

```
f(_) = 10
```

defines the function which always returns `10`, no matter what input it is given. This could also be written

```
f(n) = 10
```

but since `n` is not used, we can explicitly indicate that we do not care about it by replacing it with a wildcard pattern _.

Arithmetic patterns

An arithmetic *pattern* consists of an arithmetic expression, containing one or more variables, used as a pattern. For example,

```
f(n+2) = ...
f(2n+7) = ...
f(p/q) = ...
f(-p/(q+1)) = ...
```

are all examples of arithmetic patterns. A full description of how arithmetic patterns work would be too complex to include here; put simply, an arithmetic pattern matches whenever the input is a number of the given form. For example, `f(2n+1) = ...` matches whenever there is a number `n` such that the input is of the form `2n+1`. This is a common way to define functions depending on whether the input is even or odd:

```
f : N -> N
f(2n)   = ... n ...     -- even inputs
f(2n+1) = ... n ...     -- odd inputs
```

Note that `f(p/q)` matches whenever the input is a rational number with a numerator of `p` and a denominator of `q`. In all other cases, arithmetic patterns are generally required to have only one variable. Otherwise, the pattern is ambiguous. For example, `f(a+b) = ...` is not allowed, since `a + b = n` has many solutions for a given `n`; for a given input, there is generally no way to determine what `a` and `b` should be.

Arithmetic patterns can sometimes allow us to define things in an alternative way that does not require *subtraction*. For example, one way to define the *factorial* function is as follows:

```
fact : N -> N
fact(0)   = 1
fact(n+1) = (n+1) * fact(n)
```

In the second clause, instead of writing `fact(n) = n * fact(n .- 1)`, we can express that the clause only applies to natural numbers that are one more than another natural number.

Tuple patterns

A tuple *pattern* consists of a *tuple expression* (*i.e.* a pair, triple, ...) used as a pattern. Each component of the tuple can itself be any pattern. The simplest kind of tuple pattern would be a pair with *variables*, like

```
f : N * N -> Q
f(x,y) = ...
```

This is extremely common when defining functions that take multiple inputs. Functions that take more than two inputs can also be defined similarly:

```
f : N * Z * Z -> Z
f(a,b,c) = ...
```

(See *the page on pair types* for more details on how n-tuples work.)

The components of a pair pattern can themselves be any pattern, however, not just variables. For example,

```
f : N * N -> N
f(2n+1, 3) = 17
f(x, y) = x + y
```

The above example defines `f` to yield `17` when applied to any tuple consisting of an odd number paired with `3` (using an *arithmetic pattern* and a *literal pattern*), and `x + y` when applied to any other pair `(x,y)`.

1.6.2 Anonymous functions

Sometimes, especially when using *higher-order functions*, it is convenient to write down a *function* without having to give it a name. This can be done using a so-called "lambda expression". In its simplest form, this looks like

```
\ <var> . <expression>
```

where `<var>` stands for any variable name, and `<expression>` stands for any *expression*, which is allowed to use the variable. This represents a *function* which takes `<var>` as input, and returns `<expression>` as output.

λ (U+033B, GREEK SMALL LETTER LAMBDA) can also be used in place of `\` (a backslash is used because it looks kind of like λ).

For example, `\n. 3n+1` is the function which returns one more than three times its input.

```
Disco> (\n. 3n+1)(6)
19
```

- The thing after the lambda or backslash can actually be any *pattern*, not just a variable. For example,

  ```
  \(x,y). x + 2y
  ```

  is the function which takes a *pair* of numbers as input and returns the sum of the first number and twice the second number.

- The variable after the lambda can optionally be annotated with its type, as in `\ <var> : <type> . <expression>`. For example,

  ```
  \x:Z. x + 5
  ```

  is the function of type `Z -> Z` which returns 5 more than its input. Without the type annotation, Disco would infer `\x. x + 5` to have type `N -> N` instead.

1.6.3 Higher-order functions

A *higher-order* function is a *function* which takes other functions as input. For example:

```
twice : (N -> N) * N -> N
twice(f, x) = f(f(x))
```

The `twice` function takes a function on natural numbers `f`, along with a natural number `n`, as input, and calls `f` twice on `n`.

A *function* is an input-output relation, that is, we can think of a function as a machine, or process, that takes inputs and produces outputs according to some rule(s). For each element of the *domain*, or input type, a function specifies a single element of the *codomain*, or output type.

The *type of a function* with domain `A` and codomain `B` is written `A -> B` (or `A → B`).

Two simple examples of functions are shown below.

```
f : N -> N
f(n) = 3n + 1

g : N * N -> Q
g(x,y) = f(x) / (y - 1)
```

The function `f` takes *natural numbers* as input and produces natural numbers as output; for a given input `n` it produces the output `3n + 1`.

The function `g` takes *pairs* of natural numbers as input, and produces *rational numbers*; given the pair `(x,y)`, it produces `f(x) / (y - 1)` as output.

- Functions can be given names and defined by *pattern-matching*, as in the examples above.
- Functions can also be defined *anonymously*, using *lambda notation*. For example,

  ```
  \n. 3n + 1
  ```

  is the function which takes an input called `n` and outputs `3n + 1`. This is the same function as the example function `f` above.
- Attempting to print a function value will simply result in the type of the function being printed as a placeholder:

  ```
  Disco> (\n. 3n+1)
  < → >
  ```

  This is because once a program is running, Disco has no way in general to recover the textual definition of a function.

1.7 Collections

Disco has several built-in *collection types* (such as lists, bags, and *sets*) which represent collections of values. The pages linked below explain the different ways to create and use collections, and the operations which can be used on them.

1.7.1 Size

The *size* of a *collection* `c` (a list, bag, or *set*) can be found using the notation `|c|`. For example,

```
Disco> |{1,2,3}|
3
Disco> |{1,2,3} union {2,3,4,4}|
4
Disco> |[2 .. 7]|
6
```

1.7.2 Cartesian product

The *Cartesian product* operator, written >< (or, using Unicode, as ×), operates on two collections of the same type (either *sets*, bags, or lists), and forms the collection of all possible pairs with one element taken from the first collection and the other from the second.

- On lists, the order matters: the resulting list has the first element of the first list matched with all elements of the second list, then the second element of the first list matched with all elements of the second list, and so on.

  ```
  Disco> [2,1,1] >< [6,7]
  [(2, 6), (2, 7), (1, 6), (1, 7), (1, 6), (1, 7)]
  ```

- On sets, we simply get the set of all unique pairs.

  ```
  Disco> {2,1,1} >< {6,7}
  {(1, 6), (1, 7), (2, 6), (2, 7)}
  ```

- The behavior of Cartesian product on bags is slightly less intuitive, but follows directly from the fact that taking the Cartesian product of two lists and then converting the result to a bag always yields the same result as first converting the two lists to bags and then taking the Cartesian product (although the latter can be more efficient). That is, for all lists `l1` and `l2`, `bag(l1 >< l2) == bag(l1) >< bag(l2)`.

  In particular, if `a` is an element of bag `A` with a multiplicity of `m`, and `b` is an element of bag `B` with a multiplicity of `n`, then `(a,b)` is an element of `A >< B` with a multiplicity of `m * n`. In other words, we have `m * n` ways to form the pair `(a,b)` if we have `m` copies of `a` to choose from and `n` copies of `b` to choose from.

  ```
  Disco> bag([1,1,2,3]) >< bag([8,7,7,7])
  (1, 7) # 6, (1, 8) # 2, (2, 7) # 3, (2, 8), (3, 7) # 3, (3, 8)
  ```

1.7.3 Ellipsis

Sets and lists may be defined using *ellipsis*, that is, two or more dots meaning (intuitively) "and so on". For example:

```
Disco> {1 .. 5}
{1, 2, 3, 4, 5}
Disco> [2, 4 ... 10]
[2, 4, 6, 8, 10]
Disco> [1, 4, 9 ... 100]
[1, 4, 9, 16, 25, 36, 49]
```

Note, Disco isn't actually being all that smart here, and it won't work for any pattern at all. For example, it fails miserably to understand that we want a list of primes:

```
Disco> [2, 3, 5, 7, 11 ... 100]
[2, 3, 5, 7, 11, 22, 48, 100]
```

So what is Disco actually doing? Note first that there must always be a single number after the dots.

- If there is a single number before the dots:
  - If the first number is smaller than the last number, the resulting list or set starts at the first number and counts up by ones until reaching the last number.

```
Disco> {1 .. 5}
{1, 2, 3, 4, 5}
```

  - If the first number is greater than the last number, it counts down instead of up.

```
Disco> [10 .. 7]
[10, 9, 8, 7]
```

- If there are $k > 1$ numbers before the dots, Disco fits a $k-1$-degree polynomial to the numbers and then extends it until the next value would be greater than the value after the dots.
  - For example, for $k = 2$, this just means that Disco will extend the list using a constant gap between consecutive values (the same as the gap between the first two numbers):

```
Disco> [1, 3 .. 10]
[1, 3, 5, 7, 9]
Disco> {5, 10 .. 40}
{5, 10, 15, 20, 25, 30, 35, 40}
```

  - For $k = 3$, Disco will use a quadratic polynomial, which means we can generate things like the squares or the triangular numbers:

```
Disco> [1, 4, 9 ... 100]
[1, 4, 9, 16, 25, 36, 49]
Disco> [1, 3, 6 ... 28]
[1, 3, 6, 10, 15, 21, 28]
```

1.7.4 Set operations

The *Cartesian product* of two *finite sets* can be found using the >< operator.

```
Disco> {1,2,3} >< {'x','y'}
{(1, 'x'), (1, 'y'), (2, 'x'), (2, 'y'), (3, 'x'), (3, 'y')}
```

The *union* or *intersection* of two *finite sets* can be found using the `union` and `intersect` operators, or using the Unicode notation and .

```
Disco> {1,2,3} union {2,3,4}
{1, 2, 3, 4}
Disco> {1,2,3} intersect {2,3,4}
{2, 3}
```

The *difference* of two sets can be found using the set difference operator, written `\`:

```
Disco> {7 .. 12} \ {1 .. 10}
{11, 12}
```

You can check whether one set is a subset of another using the `subset` operator (or the Unicode symbol):

```
Disco> {2,3,4} subset {1 .. 10}
true
```

(continues on next page)

(continued from previous page)

```
Disco> {7 .. 11} subset {1 .. 10}
false
```

Note that Disco does not support the set complement operation, since the complement of a finite set is infinite whenever the domain is infinite.

1.7.5 Power set

The *power set* of a *set* is the set of all possible subsets. It can be computed using the `power` function, which takes a `Set(T)` and returns a `Set(Set(T))`:

```
Disco> power({1,2,3})
{{}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3}, {3}}
Disco> power(set("hi"))
{{}, {'h'}, {'h', 'i'}, {'i'}}
Disco> power({})
{{}}
Disco> power(power({}))
{{}, {{}}}
```

1.7.6 Comprehensions

Comprehension notation can be used to describe collections such as *sets* or lists. The general syntax for a set comprehension is

```
{ expression | qualifiers }
```

with a single expression, followed by a vertical bar `|`, followed by a list of one or more *qualifiers*. The idea is introduced through some examples below; for the precise details, see the *Details* section.

List comprehensions are similar, but use square brackets (`[`, `]`) instead of curly braces (`{`, `}`).

Examples

```
Disco> {x | x in {1..5}}   -- same as {1..5}
{1, 2, 3, 4, 5}

Disco> {3x | x in {1..5}}  -- multiply each element of {1..5} by 3
{3, 6, 9, 12, 15}

-- Pick out the elements of {1..10} that satisfy the condition
Disco> {x | x in {1 .. 10}, x^2 + 20 == 9x}
{4, 5}

-- Pick out the elements of {1..100} that satisfy all the conditions
Disco> {x | x in {1 .. 100}, x =< 10 \/ x >= 90, x mod 2 == 0}
{2, 4, 6, 8, 10, 90, 92, 94, 96, 98, 100}

-- Products of all combinations of elements from {1..4} and {1, 10, 100}
Disco> {x * y | x in {1 .. 4}, y in {1, 10, 100}}
{1, 2, 3, 4, 10, 20, 30, 40, 100, 200, 300, 400}
```

(continues on next page)

(continued from previous page)

```
-- Pairs of elements from {1..4} where the first is >= the second
Disco> {(x,y) | x in {1 .. 4}, y in {1 .. x}}
{(1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3), (4, 1), (4, 2), (4, 3), (4, 4)}
```

Details

Each *qualifier* in a comprehension can be either

- a *variable binding* of the form `<variable> in <set>`, *e.g.* `x in {1 .. 10}` or `b in {false, true}`, or
- a *guard*, which can be any *boolean* expression.

A variable binding locally defines a variable and causes it to "loop" through all the values in the given set. For example, `x in {1 .. 5}` defines the variable `x` within the comprehension, and makes `x` take on each value from 1 through 5 in turn. Multiple variable bindings will cause the loops to "nest". For example, `{ (x,y) | x in {1 .. 3}, y in {5 .. 7}}` has nine elements: `y` loops through its three possible values for *each* value of `x`.

```
Disco> { (x,y) | x in {1 .. 3}, y in {5 .. 7}}
{(1, 5), (1, 6), (1, 7), (2, 5), (2, 6), (2, 7), (3, 5), (3, 6), (3, 7)}
```

A boolean guard is checked for each combination of variable values to see if it is true. Any values of the variables which make the guard false are discarded.

Finally, any values of the variable(s) which make all the guards true are used in the expression on the left side of the `|`, and the resulting value will become an element of the set.

Putting all this together, for example, `{x^2 + y | x in {1 .. 5}, x mod 2 == 1, y in {1 .. x}, x + y > 5}` is evaluated as follows:

- `x` will loop through the values from 1 to 5.
- For each value of `x`, check whether `x mod 2 == 1`. The values which make this false (2 and 4) are discarded. The only values of `x` left are 1, 3, and 5.
- For each of the remaining values of `x`, `y` will loop through the values from 1 up to `x`.
- For each value of `y`, check whether the sum of `x` and `y` is greater than 5.
- Finally, from values of `x` and `y` which make it through both checks, we compute `x^2 + y` and put the result in the set being built.

In the end, the result is the set `{12, 26, 27, 28, 29, 30}`.

Specification

Note: In case you are curious about the precise definition and are not afraid of the details, the exact way that set comprehensions work can be defined by the following three equations, making use of the standard functions `each` and `unions`:

- `{ e | } = e`
- `{ e | x in xs, gs } = unions(each(\x. {e | gs}, xs))`
- `{ e | g, gs } = {? { e | gs } if g, {} otherwise ?}`

1.8 Error messages

There are many different error messages Disco can generate when something is wrong. Each page linked below gives more explanation and background to help you understand a particular error message.

The error messages in Disco are currently undergoing major improvements. If you have a suggestion on how a particular error message could be improved, please record it at https://github.com/disco-lang/disco/issues !

1.8.1 There is nothing named x

Something in your program, or something you typed at the prompt, refers to a variable name which is not defined.

- Did you spell the variable name correctly? (Remember that capitalization matters!)
- Did you forget to `import` a module which defines the variable you want?
- Are you trying to refer to a variable outside of the context in which it is defined? For example, a parameter to a function can only be used inside the function itself.
- Did you forget to put double quotes around a string, for example, `hello` instead of `"hello"`?

If you got this error due to something else not on the list above, please add it as a suggestion!

1.8.2 The name x is ambiguous

You tried to use a variable which has multiple definitions, so disco does not know which one you want to use. The only way this can happen is if the variable is defined in two different files. For example,

- The variable is defined both in `a.disco` and `b.disco`, and you have both `import a` and `import b` in your code.
- You have defined the variable in your own code, but it is also defined in one of the files you `import`.

The simplest solution is to rename one of the conflicting definitions. If you can't or don't want to do this, you can also make an "adapter module" to rename a variable without changing the original file. For example, suppose we have `x` defined in our own file as well as in `a.disco`. We can make a new file named `b.disco` which contains the following:

```
import a

y :: N
y = x
```

Now instead of `import a` we can say `import b`, and now we will be able to use `y` instead of `x`.

1.8.3 The definition of x must have an accompanying type signature

In Disco, you are not allowed to define a variable by simply saying `x = ...`. You must also specify the *type* of a variable by placing a *type signature* before it, like this:

```
x : N   -- a type signature for x
x = 5   -- the definition of x
```

1.8.4 The expression e must have both a blah type and also...

This error occurs sometimes when two incompatible types meet: the context in which an expression is used requires it to have a certain type, whereas the expression actually has a different type.

For example, consider the following:

```
Disco> x : N
Disco> x = 5
Disco> x(2)
Error: the expression
  x
must have both a function type and also the incompatible type
  .
```

In this example, the reason `x` must have a function type is because we applied it to an argument, like `x(2)`. The only things which can be applied to arguments are *functions*. On the other hand, we said that the type of `x` is `N`, whch is not a function.

1.8.5 Empty case expressions are not allowed

Every *case expression* must have at least one branch.

If you're getting this error, perhaps everything inside a case expression (`{? ... ?}`) is a *comment*?

1.8.6 Value did not match any of the branches in a case expression

This means that none of the conditions in a *case expression* were true. For example, consider this case expression:

```
{? 'A'  if  n < 5,
   'B'  if  n > 5
?}
```

When `n == 5` specifically, both conditions will be false, and this error will be generated.

The reason this is an error is that every *expression* must have a value; if all the conditions are false, we do not know what value the whole case expression should have.

This error may also occur when defining a *function* via *pattern matching*, if none of the patterns match a particular input. For example, consider the below definition of `f`:

```
f : N -> N
f(3) = 99
f(2n) = n
```

If we call this function on an odd input besides 3, it will generate an error, since neither of the patterns matches:

```
Disco> f(5)
Error: value did not match any of the branches in a case expression.
```

The reason the same error is generated is that internally, function definitions by cases are translated into case expressions. For example, the above definition for `f` is translated into something like

```
f : N -> N
f(m) = {? 99 if m is 3, n if m is 2n ?}
```

1.8.7 Pattern p contains duplicate variable x

A *pattern* is not allowed to contain the same variable more than once. For example, the following definition is not allowed, because the pattern `(x,x)` contains two occurrences of the variable `x`.

```
f :: N*N -> N
f(x,x) = 3
f(x,y) = 7
```

If you want to define a function which returns `3` whenever its two arguments are equal, and `7` otherwise, you could define it like this, using a *case expression*

```
f :: N*N -> N
f(x,y) = {? 3 if x == y, 7 otherwise ?}
```

1.8.8 The pattern p is supposed to have type T, but instead...

This is a similar sort of error as *The expression e must have both a blah type and also...*, but concerns *patterns* instead of *expressions*. On the one hand, disco can figure out what type a pattern should be based on the type of the *function*. On the other hand, the pattern itself may correspond to a different type. For example,

```
Disco> :{
Disco| f : N -> N
Disco| f(x,y) = x + y
Disco| :}
Error: the pattern
  (x, y)
is supposed to have type
  ,
but instead it has a pair type.
```

In this example, we have declared the type of `f` to be `N -> N`, that is, a function which takes a natural number as input and yields another natural number as output. However, we have used the pattern `(x,y)` for the input of `f`, which looks like a value of a *pair type*.

1.8.9 Duplicate type signature for x

This error message is caused by multiple *type signatures* for the same variable. It does not matter if the types are the same or different; there can only be one type signature per variable.

```
Disco> :{
Disco| x : N
Disco| x : N
Disco| :}
Error: duplicate type signature for x.
```

If this is unexpected, check that you did not misspell a variable name so it accidentally has the same name as another variable.

1.8.10 Duplicate definition for x

Just as each *variable* can *only have one type signature*, so each variable can only have one *definition*. Once a variable is defined, it is not possible to change its value to something different. This is quite different than some other languages. See the *documentation about definitions* for more information.

1.8.11 Duplicate definition for type T

This is very similar to *Duplicate definition for x*, but for type definitions instead of variables. For example,

```
Disco> type H = N * N
Disco> type H = Z + Q
Error: duplicate definition for type H.
```

1.8.12 Cyclic type definition for T

This error occurs when one or more type definitions form a cycle.

Note that recursive types, *i.e.* types defined in terms of themselves, are very much allowed (and useful)! A "cyclic type" error only occurs when a type is defined as being directly equal to itself.

For example:

```
Disco> :{
Disco| type A = B
Disco| type B = C
Disco| type C = A
Disco| :}
Error: cyclic type definition for A.
```

1.8.13 Number of arguments does not match

When defining a function, there are two ways Disco can figure out how many arguments it takes: by looking at its declared type, and by looking at the number of arguments in its definition. This error results when these are not the same.

For example, the following definition would yield this error:

```
f : N -> N -> N
f x y z = 3
```

The declared type of `f`, namely `N -> N -> N`, says that it takes two natural number inputs. However, the definition `f x y z = ...` makes it look like it takes three inputs: `x`, `y`, and `z`.

This is not a terribly informative error message, and it will likely be improved and/or split out into several separate error messages soon.

1.8.14 The type T is not searchable

When writing a property using a `forall`, we are only allowed to use types which are *searchable*, that is, types for which we can effectively generate sample values. Note this is not the same thing as being finite. For example, the type of natural numbers is searchable even though it is infinite; we can list natural numbers (either in order or randomly) in order to see which ones make the property true or false.

```
Disco> :test forall x:N. x < 10    -- this works as expected
  - Test is false: x. x < 10
    Counterexample:
      x = 10
```

On the other hand, the *function* type *N -> N* is not searchable: there is no way to list all functions of type *N -> N*. The below property is in fact false, but Disco can't handle it:

```
Disco> :test forall f : N -> N. f(4) > 6
Error: the type
   →
is not searchable (i.e. it cannot be used in a forall).
```

1.8.15 There is no built-in or user-defined type named X

This is similar to *There is nothing named x*, but with types. You have referred to a type that does not exist.

- Did you spell the name of the type correctly? (Remember that capitalization matters!)
- Did you forget to `import` a module which defines the type you want?

1.8.16 Wildcards are not allowed in expressions

A *wildcard pattern*, written using an underscore (_), can be used in a pattern (on the *left* side of an equals sign) to indicate that you don't care about a certain value.

However, you are not allowed to use a wildcard anywhere on the *right* side of an equals sign. If you promise to give me two books you can't just give me one and say you don't care about the other one; likewise, if you promise to deliver (say) a pair of natural numbers, you not allowed to say you don't care what one of them is:

```
f : Char -> N * N
f(_) = (_, 3)
```

The first _ is fine: the function `f` doesn't need to care what `Char` input it is given. But the second _ is not OK: `f` has promised to return a pair of natural numbers, and it had better fulfill its promise.

1.8.17 Not enough/too many arguments for the type T

Some built-in types expect to be given one or more types as arguments (for example, *List* and *Set* both expect one argument; *Map* expects two). You can also define your own types that expect arguments. These error messages show up when you have given the wrong number of arguments to a type. For example:

```
Disco> t : List
Error: not enough arguments for the type 'List'.
Disco> t : List(N,Q)
Error: too many arguments for the type 'List'.
Disco> type MyType(a,b,c) = List(a) * Set(b) * List(c)
Disco> q : MyType(Char,Bool)
Error: not enough arguments for the type 'MyType'.
```

1.8.18 Unknown type variable

This error always refers to a type definition, which uses a type variable that was not a parameter of the type being defined. For example:

```
Disco> type T(a,b) = N * c
Error: Unknown type variable 'c'.
```

In this example, we are defining the type `T` which has parameters `a` and `b`. We are thus allowed to use `a` and `b` anywhere inside the definition of `T`. However, here we use `c`, which is not defined.

- Did you misspell a variable name?
- Did you forget to add the variable as a parameter of the type? For example, if we want to define a type of parameterized trees, but write `type T = Unit + a * T * T`, we would get this error; what we should write insitead is `type T(a) = Unit + a * T(a) * T(a)`.

1.8.19 Recursive occurrences of T may only have type variables as arguments

For technical reasons, when defining a parameterized type, any recursive occurrences of the type in its own definition can only have type variables as arguments. For example, this is perfectly OK:

```
type T(a) = Unit + a * T(a) * T(a)
```

Notice how every occurrence of `T` on the right-hand side of the = has the variable `a` as an argument.

Even this is OK:

```
type Alt(a,b) = Unit + a * Alt(b,a)
```

In this example, the `Alt` on the right-hand side of the = has its arguments in the opposite order from the one on the left-hand side, but that is OK as long as they are all type variables.

These examples, on the other hand, are not OK:

```
type Bad1(a) = Unit + Bad1(N)
type Bad2(a) = Unit + Bad2(Bad2(a))
```

1.8.20 The shape of two types does not match

This is a really horrible, uninformative error message; I'm sorry! I hope to improve it soon.

In the meantime, this error is generally caused by the types of different things not lining up. For example:

```
h : Z*Z -> Bool
h(3) = true
```

In this example, the type of `h` says that `h` takes a pair of integers as input. However, the definition of `h` looks like it takes a single natural number as input, and these types don't match.

Some tips for pinpointing the error:

- Try narrowing down the source of the error by checking small pieces of code by themselves.
- Try asking Disco for the types of various pieces at the prompt to see if they match what you think the type should be. For example, Disco can tell us that the type of `true` is `Bool` (which matches the desired output type), but the type of `3` is `N` (which does not match).
- Ask for help if you are stuck!

1.8.21 Typechecking failed

This error message is the absolute worst. It is used in any and all situations where something went wrong and Disco couldn't make sense of the types in your program.

1.8.22 Values of type T cannot be...

This error message comes up when you are trying to apply certain operations to types that do not allow that operation. For example:

- Trying to *subtract* or *divide* two *natural numbers*
- Trying to *compare* two functions

1.8.23 Type variable x represents any type, so we cannot assume...

A *polymorphic function* has to be able to work for *any* input type. Thus, it cannot assume that input values of a polymorphic type support any operations in particular. For example, the type of the function `h` below claims it works for any type `a` at all, but the implementation of `h` uses subtraction (which does not actually work for any type):

```
h : a -> a
h(x) = x - 3
```

1.9 Symbols

The following table shows all the fancy Unicode symbols that can be used in Disco. The first column shows the symbol itself (suitable for copy-pasting). For each symbol, the table shows the corresponding Unicode codepoint, the LaTeX command that can be used to generate the symbol, an equivalent ASCII representation that can be used in Disco to avoid copy-pasting a fancy symbol, and the meaning of the symbol with a link to the relevant Disco documentation.

Symbol	Codepoint	LaTeX	ASCII equivalent	Meaning + documentation
¬	U+AC	`\neg`	`not`	*Not*
	U+2227	`\land`	`/\`	*And*
	U+2227	`\lor`	`\/`	*Or*
→	U+2192	`\to`	`->`	*Implies*; *function type*
	U+2192	`\iff`	`<->`	*If and only if*
	U+2260	`\neq`	`/=`, `!=`	*Not equal to*
	U+2264	`\leq`	`<=`, `=<`	*Less-than-or-equal-to*
	U+2265	`\geq`	`>=`, `=>`	*Greater-than-or-equal-to*
	U+2200	`\forall`	`forall`	*Universal quantification*
	U+2203	`\exists`	`exists`	*Existential quantification*
	U+2238		`.-`	*Saturating subtraction*
	U+2208	`\in`	`elem`	*Element of*
	U+2286	`\subseteq`	`subset`	*Subset of*
	U+222A	`\cup`	`union`	*Set union*
	U+2229	`\cap`	`intersect`	*Set intersection*
	U+2A2F	`\times`	`><`	*Cartesian product*; *pair type*
	U+228E	`\uplus`	`+`	*Sum type*
	U+2115	`\mathbb{N}`	`N`	*Natural numbers*
	U+2124	`\mathbb{Z}`	`Z`	*Integers*
	U+1D53D	`\mathbb{F}`	`F`	*Fractional numbers*
	U+211A	`\mathbb{Q}`	`Q`	*Rational numbers*
λ	U+033B	`\lambda`	`\`	*Anonymous function*
	U+25A0	`\blacksquare`	`unit`	*Unit value*

CHAPTER 2

Quick Tutorial for experienced functional programmers

Disco is a small yet expressive, pure functional programming language designed especially to be used in the context of a discrete mathematics course. Right now it is in a rough prototype stage; this tutorial is only useful for developers who already have some knowledge of functional programming.

2.1 Getting started

After you have been added as a collaborator to the disco github repository, get the source code via SSH:

```
git clone git@github.com:disco-lang/disco.git
```

If you are not a collaborator on the repository you can also get the source code via HTTPS:

```
git clone https://github.com/disco-lang/disco.git
```

Make sure you have the stack tool installed. Then navigate to the root directory of the disco repository, and execute

```
stack setup
stack build --fast
```

(This may take quite a while the first time, while `stack` downloads and builds all the dependencies of `disco`.)

After building disco with `stack build`, to run the disco REPL (Read-Eval-Print Loop), type `stack exec disco` at a command prompt. You should see a disco prompt that looks like this:

```
Disco>
```

To run the test suite, you can execute

```
stack test --fast
```

See the `build` script in the root of the repository for an example of additional arguments you may wish to pass to `stack build` for a tight edit-compile-test loop while working on the code.

2.2 Arithmetic

As a computational platform for learning discrete mathematics, at the core of disco is of course the ability to compute with numbers. However, unlike many other languages, disco does not support real (*aka* floating-point) numbers at all—they are not typically needed for discrete mathematics, and omitting them simplifies the language quite a bit. To compensate, however, disco has sophisticated built-in support for rational numbers.

2.2.1 Basic arithmetic

To start out, you can use disco as a simple calculator. For example, try entering the following expressions, or others like them, at the `Disco>` prompt:

- `2 + 5`
- `5 - 8`
- `5 * (-2)`
- `(1 + 2)(3 + 4)`
- `2 ^ 5`
- `2 ^ 5000`
- `4 .- 2`
- `2 .- 4`

The last two expressions use the saturating subtraction operator, `.-`, which takes two numeric operands, a and b, and returns $a - b$ if $a > b$, and 0 otherwise. Note that unlike regular subtraction, the result of a saturating subtraction will always be a natural number.

Also notice that it is not always necessary to write $*$ for multiplication: as is standard mathematical notation, we may often omit it, as in `(1 + 2)(3 + 4)`, which means the same as `(1 + 2) * (3 + 4)`. (For precise details on when the asterisk may be omitted, see the discussion in the section on functions.) Notice also that integers in disco may be arbitrarily large.

Now try these:

- `3/7 + 2/5`
- `2 ^ (-5)`

The results may come as a bit of a surprise if you are already used to other languages such as Java or Python, which would yield a floating-point (*i.e.* real) number; as mentioned before, disco does not support floating-point numbers. However, rational numbers can still be entered using decimal notation. Try these expressions as well:

- `2.3 + 1.6`
- `1/5.0`
- `1/7.0`

Disco automatically picks either fractional or decimal notation for the output, depending on whether any values with decimal points were used in the input (for example, compare `1/5` and `1/5.0`, or `1.0/5`). Note that `1/7.0` results in `0.[142857]`; can you figure out what the brackets indicate?

The standard `floor` and `ceiling` operations are built-in:

```
Disco> floor (17/3)
5
Disco> ceiling (17/3)
6
```

Just for fun, disco also supports standard mathematical notation for these operations via Unicode characters:

```
Disco>  17/3
5
Disco>  17/3
6
```

Integer division, which rounds down to the nearest integer, can be expressed using `//`:

```
Disco> 5 // 2
2
Disco> (-5) // 2
-3
```

`x // y` is always equivalent to `floor (x/y)`, but is provided as a separate operator for convenience.

The counterpart to integer division is `mod`, which gives the remainder when the first number is divided by the second:

```
Disco> 5 mod 2
2
Disco> (2^32) mod 7
4
Disco> (2^32) % 7
```

The `%` operator may also be used as a synonym for `mod`.

Finally, the `abs` function is provided for computing absolute value:

```
Disco> abs 5
5
Disco> abs (-5)
5
```

2.2.2 Advanced arithmetic

Disco also provides a few more advanced arithmetic operators which you might not find built in to other languages.

- The `divides` operator can be used to test whether one number evenly divides another. Try evaluating these expressions:
  - `2 divides 20`
  - `2 divides 21`
  - `(-2) divides 20`
  - `2 divides (-20)`
  - `7 divides (2^32 - 4)`
  - `(1/2) divides (3/2)`
  - `(1/5) divides (3/2)`
  - `1 divides 10`

  - `0 divides 10`
  - `10 divides 0`
  - `0 divides 0`

  The last three expressions may be surprising, but follow directly from the definition: `a divides b` is true if there is an integer `k` such that `a*k = b`. For example, there is no `k` such that `0*k = 10`, so `0 divides 10` is false.

  Note that a vertical line is often used to denote divisibility, as in $3 \mid 21$, but disco does not support this notation, since the vertical line is used for other things (and besides, it is typically not a good idea to use a visually symmetric operator for a nonsymmetric relation).

- The `choose` operator can be used to compute binomial coefficients. For example, `5 choose 2` is the number of ways to select two things out of five.
- The factorial function is available via standard mathematical notation:

```
Disco> 20!
2432902008176640000
```

- A square root (`sqrt`) function is provided which rounds the result down to the nearest integer (remember that disco does not support arbitrary real numbers).

```
Disco> sqrt (299^2 + 1)
299
Disco> sqrt (299^2 .- 1)
298
```

2.3 Types

Every value in disco has a *type*. Types play a central role in the language, and help guide and constrain programs. All the types in a program must match correctly (*i.e.* the program must *typecheck*) before it can be run. The type system has for the most part been designed to correspond to common mathematical practice, so if you are used to type systems in other programming languages (even other functional languages) you may be in for a surprise or two.

Disco can often infer the type of an expression. To find out what type disco has inferred for a given expression, you can use the `:type` command. For example:

```
Disco> :type 3
3 :
Disco> :type 2/3
2 / 3 :
Disco> :type [1,2,5]
[1, 2, 5] : List
```

The colon in `3 :` can be read "has type" or "is a", as in "three is a natural number". A colon can also be used to give an explicit type to an expression, for example, when you want to specify a type other than what disco would infer. For example:

```
Disco> :type 3 + 5
3 + 5 :
Disco> :type (3 : Integer) + 5
(3 : ) + 5 :
```

The above example shows that normally, disco infers the type of `3 + 5` to be a natural number, but we can force the `3` to be treated as an `Integer`, which in turn forces the whole expression to be inferred as an integer.

2.3.1 Primitive numeric types

Disco has four built-in primitive numeric types: natural numbers, integers, fractions (*i.e.* nonnegative rationals), and rationals.

- The type of natural numbers, written `Natural`, `Nat`, `N`, or , includes the counting numbers $0, 1, 2, \ldots$.
- The type of integers, written `Integer`, `Int`, `Z`, or , includes the natural numbers as well as their negatives.
- The type of fractions (*i.e.* nonnegative rationals), written `Fractional`, `Frac`, `F`, or , includes all ratios of the form a/b where a and b are natural numbers, with $b \neq 0$.
- The type of rational numbers, written `Rational`, `Q` or , includes all ratios of integers.

In mathematics, it is typically not so common to think of the nonnegative rationals $\mathbb{F}$ as a separate set by themselves; but this is mostly for historical reasons and because of the way the development of rational numbers is usually presented. The natural numbers support addition and multiplication. Extending them to support subtraction yields the integers; then, extending these again to support division yields the rationals. However, what if we do these extensions in the opposite order? Extending the natural numbers to support division results in the positive rational numbers; then extending these with subtraction again yields the rationals. All told, the relationship between these four types forms a diamond-shaped lattice:

```
  Q
 / \
Z   F
 \ /
  N
```

Each type is a subset of the type or types above it. Going northwest in this diagram ($\mathbb{N} \to \mathbb{Z}$ or $\mathbb{F} \to \mathbb{Q}$) corresponds to allowing negatives, that is, subtraction; going northeast ($\mathbb{N} \to \mathbb{F}$ or $\mathbb{Z} \to \mathbb{Q}$) corresponds to allowing reciprocals, that is, division.

Try evaluating each of the following expressions at the disco prompt, and also request their inferred type with the `:type` command. What type does disco infer for each? Why?

- `1 + 2`
- `3 * 7`
- `1 - 2`
- `1 / 2`
- `(1 - 2) / 3`

Going southeast in the lattice (getting rid of negatives) is accomplished with the absolute value function `abs`. Going southwest (getting rid of fractions) is accomplished with `floor` and `ceiling`.

Note that disco supports *subtyping*, that is, values of type `S` can be automatically "upgraded" to another type `T` as long as `S` is a "subtype" (think: subset) of `T`. For example, a natural number can be automatically upgraded to an integer.

```
Disco> (-1 : Z) + (3 : N)
2
Disco> :type (-1 : Z) + (3: N)
(-1 : ) + (3 : ) :
```

In the above example, the natural number `3` is automatically upgraded to an integer so that it can be added to `-1`. When we discuss functions later, we will see that this principle extends to function arguments as well: for example, if a function is expecting an integer as input, it is acceptable to give it a natural number, since the natural number can be upgraded to an integer.

2.3.2 Other types

There are many other types built into disco as well—`Bool`, `Void`, `Unit`, `List`, product, and sum types, to name a few. These will be covered throughout the rest of the tutorial in appropriate places. For now, try executing these commands and see if you can guess what is going on:

- `:type false`
- `:type unit`
- `:type [1, 2, 3]`
- `:type [1, 2, -3]`
- `:type [1, 2, -3, 4/5]`
- `:type [[1,2], [3,4,5]]`
- `:type (1, true)`
- `:type left(3)`

2.4 Disco files and the disco REPL

For anything beyond simple one-off calculations that can be entered at the disco prompt, disco definitions may be stored in a file which can be loaded into the REPL.

2.4.1 Disco files

Disco files typically end in `.disco`. Here is a simple example:

Listing 1: example/basics.disco

```
approx_pi : Rational
approx_pi = 22/7

increment : N -> N
increment(n) = n + 1
```

This file contains definitions for `approx_pi` and `increment`. Each definition consists of a *type signature* of the form `<name> : <type>`, followed by an equality of the form `<name> = <expression>`. Both parts of a definition are required; in particular, if you omit a type signature, disco will complain that the name is not defined. The example file shown above contains two definitions: `approx_pi` is defined to be the `Rational` number $22/7$, and `increment` is defined to be the function which outputs one more than its natural number input. (Functions and the syntax for defining them will be covered in much more detail in an upcoming section of the tutorial.)

The order of definitions in a `.disco` file does not matter; each definition may refer to any other definition in the whole file.

To load the definitions in a file into the disco REPL, you can use the `:load` command. After successfully loading a file, all the names defined in the file are available for use; the `:names` command can be used to list all the available names. For example:

```
Disco> :load example/basics.disco
Loading example/basics.disco...
Loaded.
Disco> :names
```

(continues on next page)

(continued from previous page)

```
approx_pi :
increment :  →
Disco> approx_pi
22/7
Disco> increment(3)
4
Disco> :type increment
increment :  →
Disco> approx_pi + increment(17)
148/7
```

(If you want to follow along, note that the above interaction assumes that the disco REPL was run from the *docs/tutorial* subdirectory.)

2.4.2 Comments and documentation

Comments in disco have a similar syntax to Haskell, with the exception that only single-line comments are supported, and not multi-line comments. In particular, two consecutive hyphens `--` will cause disco to ignore everything until the next newline character.

Listing 2: example/comment.disco

```
-- This is a comment
approx_pi : Rational
approx_pi = 22/7   -- an OK approximation

-- The following function is very complicated
-- and took about three weeks to write.
-- Don't laugh.
increment : N -> N
increment(n) = n + 1 -- one more than the input
```

Comments can be placed anywhere and are literally ignored by disco. In many cases, however, the purpose of a comment is to provide documentation for a function. In this case, disco supports special syntax for *documentation*, which must be placed before the type signature of a definition. Each line of documentation must begin with `|||` (three vertical bars).

Listing 3: example/doc.disco

```
||| A reasonable approximation of pi.
approx_pi : Rational
approx_pi = 22/7   -- an OK approximation

||| Take a natural number as input, and return the natural
||| number which is one greater.
|||
||| Should not be used while operating heavy machinery.
-- This comment will be ignored.
increment : N -> N
increment(n) = n + 1

fizz : N
fizz = 1
```

When this file is loaded into the disco REPL, we can use the `:doc` command to see the documentation associated

with each name.

```
Disco> :load example/doc.disco
Loading example/doc.disco...
Loaded.
Disco> :doc approx_pi
approx_pi :

A reasonable approximation of pi.

Disco> :doc increment
increment :  →

Take a natural number as input, and return the natural
number which is one greater.

Should not be used while operating heavy machinery.

Disco> :doc fizz
fizz :
```

Since `fizz` does not have any associated documentation, the `:doc` command simply shows its type.

2.4.3 Other REPL commands

The disco REPL has a few other commands which are useful for disco developers.

- `:parse` shows the fully parsed form of an expression.

```
Disco> :parse 2 + [3,4 : Int]
TBin_ () Add (TNat_ () 2) (TContainer_ () ListContainer [(TNat_ () 3,
↪Nothing),(TAscr_ () (TNat_ () 4) (Forall (<[]> TyAtom (ABase Z))),
↪Nothing)] Nothing)
```

- `:pretty` shows the pretty-printed form of a term (without typechecking it).

```
Disco> :pretty 2 + [3,4:Int]
2 + [3, (4 : )]
```

- `:desugar` shows the desugared term corresponding to an expression.

```
Disco> :desugar [3,4]
3 :: 4 :: []
```

- `:compile` shows the compiled core language term corresponding to an expression.

```
Disco> :compile [3 - 4]
CCons 1 [CApp (CConst OAdd) [(Lazy,CCons 0 [CNum Fraction (3 % 1),CApp␣
↪(CConst ONeg) [(Strict,CNum Fraction (4 % 1))]])],CCons 0 []]
```

2.5 Logic

2.5.1 Booleans

The type of booleans, written `Bool` or `Boolean`, represents logical truth and falsehood. The two values of this type are written `true` and `false`. (For convenience `True` and `False` also work.)

- Logical AND can be written `and`, `&&`, or ∧ (note that ∧ is `U+2227 LOGICAL AND`, not a caret symbol `^`, which is reserved for exponentiation).
- Logical OR is written `or`, `||`, or ∨ (`U+2228 LOGICAL OR`).
- Logical negation (NOT) is written `not` or ¬ (`U+00AC NOT SIGN`).
- Logical implication is written `implies` or `==>`.

```
Disco> true and false
false
Disco> true || false
true
Disco> not (true  true)
false
Disco> ¬ (false or false or false or true)
false
Disco> true implies false
false
Disco> false implies true
true
```

2.5.2 Equality testing

If you have two disco values of the same type, in almost all cases you can compare them to see whether they are equal using ==, resulting in a `Bool` value.

```
Disco> 2 == 5
false
Disco> 3 * 7 == 2*10 + 1
true
Disco> (3/5)^2 + (4/5)^2 == 1
true
Disco> false == False
true
```

The /= operator tests whether two values are *not* equal; it is just the logical negation of ==.

2.5.3 Comparison

Again, in almost all cases values can be compared to see which is less or greater, using operators <, <=, >, or >=.

```
Disco> 2 < 5
true
Disco> false < true
true
```

Comparisons can also be chained; the result is obtained by comparing each pair of values according to the comparison between them, and taking the logical AND of all the results. For example:

```
Disco> 1 < 3 < 8 < 99
true
Disco> 2.2 < 5.9 > 3.7 < 8.8 > 1.0 < 9
true
Disco> x : Int
Disco> x = 5
Disco> 2 < x < 10
true
```

2.6 Structural types

In addition to the primitive types covered so far, disco also has sum and product types which can be used to build up more complex structures out of simpler ones.

2.6.1 Product types

The product of two types, written using an asterisk `*` or Unicode times symbol × (`U+00d7 MULTIPLICATION SIGN`), is a type whose values are ordered pairs of values of the component types. Pairs are written using standard ordered pair notation.

Listing 4: example/pair.disco

```
pair1 : N * Q
pair1 = (3, -5/6)

pair2 : Z × Bool
pair2 = (17 + 22, (3,5) < (4,2))

pair3 : Bool * (Bool * Bool)
pair3 = (true, (false, true))

pair4 : Bool * Bool * Bool
pair4 = (true, false, true)
```

`pair1` in the example above has type `N * Q`, that is, the type of pairs of a natural number and a rational number; it is defined to be the pair containing `3` and $-5/6$. `pair2` has type `Z × Bool` (using the alternate syntax × in place of `*`), and contains two values: $17 + 22$, and the result of asking whether $(3, 5) < (4, 2)$.

```
Disco> pair2
(39, true)
```

Pairs are compared lexicographically, which intuitively means that the first component is most important, the second component breaks ties in the first component, and so on. For example, $(a, b) < (c, d)$ if either $a < c$ (in which case b and d don't matter) or if $a = c$ and $b < d$. This is why `(3,5) < (4,2)` evaluates to `true`. Of course, two pairs are equal exactly when their first elements are equal and their second elements are equal.

`pair3` shows that pairs can be nested: it is a pair whose second component is also a pair. `pair4` looks like an ordered triple, but in fact we can check that `pair3` and `pair4` are equal!

```
Disco> pair3 == pair4
true
```

Really, `pair4` is just syntax sugar for `pair3`. In general:

- The type `X * Y * Z` is interpreted as `X * (Y * Z)`.
- The tuple `(x,y,z)` is interpreted as `(x,(y,z))`.

This continues recursively, so, for example, `A * B * C * D * E` means `A * (B * (C * (D * E)))`. Put another way, disco really only has pairs, but appears to support arbitrarily large tuples by encoding them as right-nested pairs.

If you want *left*-nested pairs you can use explicit parentheses: for example, `(Bool * Bool) * Bool` is not the same as `Bool * Bool * Bool`, and has values such as `((false, true), true)`.

2.6.2 Sum types

If `X` and `Y` are types, their *sum*, written `X + Y` (or `X ⅀ Y`, using `U+228e MULTISET UNION`), is the disjoint union of `X` and `Y`. That is, values of type `X + Y` are either values of `X` or values of `Y`, along with a "tag" so that we know which it is. The possible tags are `left` and `right` (to indicate the type on the left or right of the +). For example:

Listing 5: example/sum.disco

```
sum1 : N + Bool
sum1 = left(3)

sum2 : N + Bool
sum2 = right(false)

sum3 : N + N + N
sum3 = right(right(3))
```

`sum1` and `sum2` have the same type, namely, `N + Bool`; values of this type consist of either a natural number or a boolean. `sum1` contains a natural number, tagged with `left`; `sum2` contains a boolean tagged with `right`.

Notice that `X + X` is a different type than `X`, because we get two distinct copies of all the values in `X`, some tagged with `left` and some with `right`. This is why we call a sum type a *disjoint* union.

Iterated sum types, as in `sum3`, are handled in exactly the same way as iterated product types: `N + N + N` is really syntax sugar for `N + (N + N)`. `sum3` therefore begins with a `right` tag, to show that it contains a value of the right-hand type, namely, `N + N`; this value in turn consists of another `right` tag along with a value of type `N`. Other values of the same type `N + N + N` include `right(left(6))` and `left(5)`.

2.6.3 Unit and Void types

Disco has two other special built-in types which are rarely useful on their own, but often play an important role in describing other types.

- The type `Unit` has just a single value, called `unit` or .

  ```
  Disco> :type unit
   : Unit
  ```

- The type `Void` has *no* values.

2.6.4 Counting and enumerating types

For any type which has only a finite number of values, disco can count how many values there are, using the `count` operator, or list them using `enumerate` (we will learn more about lists later in the tutorial).

```
Disco> count ((Bool * (Bool + Bool)) + Bool)
right 10
Disco> enumerate ((Bool * (Bool + Bool)) + Bool)
[left (false, left false), left (false, left true), left (false, right false),
 left (false, right true), left (true, left false), left (true, left true),
 left (true, right false), left (true, right true), right false, right true]
Disco> enumerate (Bool * Bool * Bool)
[(false, false, false), (false, false, true), (false, true, false), (false, true,␣
↪true),
 (true, false, false), (true, false, true), (true, true, false), (true, true, true)]
```

2.7 Functions

The type of functions with input `X` and output `Y` is written `X -> Y`. Some basic examples of function definitions are shown below.

Listing 6: example/function.disco

```
f : N -> N
f(x) = x + 7

g : Z -> Bool
g(n) = (n - 3) > 7

factorial : N -> N
factorial(0) = 1
factorial(n) = n * factorial(n .- 1)
```

- The function `f` takes a natural number as input, and returns the natural number which is 7 greater. Notice that `f` is defined using the syntax `f(x) = ...`. In fact, the basic syntax for function arguments is juxtaposition, just as in Haskell; the syntax `f x = ...` would work as well. Stylistically, however, `f(x) = ...` is to be preferred, since it matches standard mathematical notation.
- The function `g` takes an integer `n` as input, and returns a boolean indicating whether `n - 3` is greater than 7. Note that this function cannot be given the type `N -> Bool`, since it uses subtraction.
- The recursive function `factorial` computes the factorial of its input. Top-level functions such as `factorial` are allowed to be recursive. Notice also that `factorial` is defined by two cases, which are matched in order from top to bottom, just as in Haskell.

Functions can be given inputs using the same syntax:

```
Disco> f(2^5)
39
Disco> g(-5)
false
Disco> factorial(5 + 6)
39916800
```

"Multi-argument functions" can be written as functions which take a product type as input. (This is again a stylistic choice: disco certainly supports curried functions as well. But in either case, disco fundamentally supports only one-argument functions.) For example:

Listing 7: example/multi-arg-functions.disco

```
gcd : N * N -> N
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)

discrim : Q * Q * Q -> Q
discrim(a,b,c) = b^2 - 4*a*c

manhattan : (Q*Q) * (Q*Q) -> Q
manhattan ((x1,y1), (x2,y2)) = abs (x1-x2) + abs (y1-y2)
```

All of these examples are in fact *pattern-matching* on their arguments, although this is most noticeable with the last example, which decomposes its input into a pair of pairs and gives a name to each component.

Functions in disco are first-class, and can be provided as input to another function or output from a function, stored in data structures, *etc*. For example, here is how one could write a higher-order function to take a function on natural numbers and produce a new function which iterates the original function three times:

Listing 8: example/higher-order.disco

```
thrice : (N -> N) -> (N -> N)
thrice(f)(n) = f(f(f(n)))
```

2.7.1 Anonymous functions

The syntax for an anonymous function in disco consists of a *lambda* (either a backslash or an actual λ) followed by a pattern, a period, and an arbitrary disco expression (the *body*).

The pattern can be a single variable name or a more complex pattern. Note that patterns can also contain type annotations. Unlike in, say, Haskell, there is no special syntactic sugar for curried multi-argument functions; one can just write nested lambdas.

Here are a few examples to illustrate the possibilities:

```
Disco> thrice(\x. x*2)(1)
8
Disco> thrice(\z:Nat. z^2 + 2z + 1)(7)
17859076
Disco> (\(x,y). x + y) (3,2)
5
Disco> (\x:N. \y:Q. x > y) 5 (9/2)
true
```

2.7.2 Let expressions

Let expressions are a mechanism for defining new variables for local use within an expression. For example, `3 + (let y = 2 in y + y)` evaluates to `7`: the expression `y + y` is evaluated in a context where `y` is defined to be `2`, and the result is then added to `3`. The simplest syntax for a let expression, as in this example, is `let <variable> = <expression1> in <expression2>`. The value of the let expression is the value of `<expression2>`, which may contain occurrences of the `<variable>`; any such occurrences will take on the value of `<expression1>`.

More generally:

- A `let` may have multiple variables defined before `in`, separated by commas.
- Each variable may optionally have a type annotation.
- The definitions of later variables may refer to previously defined variables.
- However, the definition of a variable in a `let` may not refer to itself; only top-level definitions may be recursive.

Here is a (somewhat contrived) example which demonstrates all these features:

Listing 9: example/let.disco

```
f : Nat -> List(Nat)
f n =
  let x : Nat = n//2,
      y : Nat = x + 3,
      z : List(Nat) = [3,x,y]
  in  n :: z
```

An important thing to note is that a given definition in a `let` expression will only ever be evaluated (at most) once, even if the variable is used multiple times. `let` expressions are thus a way for the programmer to ensure that the result of some computation is shared. `let x = e in f x x` and `f e e` will always yield the same result, but the former might be more efficient, if `e` is expensive to calculate.

2.7.3 Disambiguating function application and multiplication

As previously mentioned, the fundamental syntax for applying a function to an argument is *juxtaposition*, that is, simply putting the function next to its argument (with a space in between if necessary).

However, disco also allows multiplication to be written in this way. How can it tell the difference? Given an expression of the form `X Y` (where `X` and `Y` may themselves be complex expressions), disco uses simple *syntactic* rules to distinguish between multiplication and function application. In particular, note that the *types* of `X` and `Y` do not enter into it at all (it would greatly complicate matters if parsing and typechecking had to be interleaved—even though this is what human mathematicians do in their heads; see the discussion below).

To decide whether `X Y` is function application or multiplication, disco looks only at the syntax of `X`; `X Y` is multiplication if and only if `X` is a *multiplicative term*, and function application otherwise. A multiplicative term is one that looks like either a natural number literal, or a unary or binary operation (possibly in parentheses). For example, `3`, `(-2)`, and `(x + 5)` are all multiplicative terms, so `3x`, `(-2)x`, and `(x + 5)x` all get parsed as multiplication. On the other hand, an expression like `(x y)` is always parsed as function application, even if x and y both turn out to have numeric types; a bare variable like `x` does not count as a multiplicative term. Likewise, `(x y) z` is parsed as function application, since `(x y)` is not a multiplicative term.

Note: You may enjoy reflecting on how a *human* mathematician does this disambiguation. In fact, they are doing something much more sophisticated than disco, implicitly using information about types and social conventions regarding variable names in addition to syntactic cues. For example, consider $x(y+3)$ versus $f(y+3)$. Most mathematicians would unconsciously interpret the first as multiplication and the second as function application, due to standard conventions about the use of variable names x and f. On the other hand, in the sentence "Let x be the function which doubles an integer, and consider $v = x(y+3)$", any mathematician would have no trouble identifying this use of $x(y+3)$ as function application, although they might also rightly complain that x is a strange choice for the name of a function.

2.7.4 Operator functions

Operators can be manipulated as functions using the ~ notation. The tilde goes wherever the argument to the operator would go. This can be used, for example, to pass an operator to a higher-order function.

```
Disco> :type ~+~
~+~ :  x  →

Disco> import list
Loading list.disco...
Disco> foldr(~+~,0,[1 .. 10])
55

Disco> -- factorial
Disco> :type ~!
~! :  →

Disco> -- negation
Disco> :type -~
-~ :  →
```

2.8 Case expressions

Fundamentally, the only construct available in disco which allows choosing between multiple alternatives is case analysis using a *case expression*. (The other is multi-clause functions defined via pattern-matching, but in fact that is really only syntax sugar for a case expression.)

The syntax of case expressions is inspired by mathematical notation such as

$$f(x) = \begin{cases} x + 2 & x < 0 \\ x^2 - 3x + 2 & 0 \leq x < 10 \\ 5 - x & \text{otherwise} \end{cases}$$

Here is how one would write a corresponding definition in disco:

Listing 10: example/case.disco

```
f : Z -> Z
f(x) = {? x + 2          if x < 0,
          x^2 - 3x + 2   if 0 <= x < 10,
          5 - x          otherwise
       ?}
```

The entire expression is surrounded by `{? ... ?}`; the curly braces are reminiscent of the big brace following $f(x) = \ldots$ in the standard mathematical notation, but we don't want to use plain curly braces (since those will be used for sets), so question marks are added (which are supposed to be reminiscent of the fact that case expressions are about asking questions).

2.8.1 Case syntax and semantics

More formally, the syntax of a case expression consists of one or more *branches*, separated by commas, enclosed in `{? ... ?}`. (Whitespace, indentation, *etc.* formally does not matter, though something like the style shown in the example above is encouraged.)

Each *branch* consists of an arbitrary expression followed by zero or more *guards*. When a case expression is evaluated, each branch is tried in turn; the first branch which has *all* its guards succeed is chosen, and the value of its expression becomes the value of the entire case expression. In the example above, this means that first `x < 0` is evaluated; if it is true then `x + 2` is chosen as the value of the entire case expression (and the rest of the branches are ignored). Otherwise, `0 <= x < 10` is evaluated; and so on.

There are three types of guards:

- A *boolean guard* has the form `if <expr>` or `when <expr>`, where `<expr>` is an expression of type `Bool`. It succeeds if the expression evaluates to `true`. There is no difference between `if` and `when`; they are simply synonyms.
- A *pattern guard* has the form `if <expr> is <pattern>`, or `when <expr> is <pattern>`. It succeeds if the expression `<expr>` matches the pattern `<pattern>`.
- For convenience, the special guard `otherwise` is equivalent to `if true`.

Here is an example using both boolean and pattern guards:

Listing 11: example/case-pattern.disco

```
g : Z*Z -> Z
g(p) = {? 0      when p is (3,_),
          x + y  when p is (x,y) when x > 5 or y > 20,
          -100   otherwise
       ?}
```

Here is the result of evaluating `g` on a few example inputs:

```
Disco> g(3,9)
0
Disco> g(4,3)
-100
Disco> g(16,15)
31
```

When a pattern containing variables matches, the variables are bound to the corresponding values, and are in scope in both the branch expression as well as any subsequent guards. In the example above, when the pattern `(x,y)` matches `p`, both `x` and `y` may be used in the branch expression (`x + y` in this case) as well as in the second guard `if x > 5 or y > 20`. That is, the guards in this branch will only succeed if `p` is of the form `(x,y)` and either `x > 5` or `y > 20`, in which case the value of the whole case expression becomes the value of `x + y`; for example, `g(16,15) = 31`.

Warning: Be careful not to get a Boolean guard using == confused with a pattern guard using `is`. The difference is in how variables are handled: boolean guards can only use existing variables; pattern guards create new variables. For example, `... when p is (x,y)` matches a tuple `p` and gives the names `x` and `y` to the components. On the other hand, `... if p == (x,y)` will probably complain that `x` and `y` are undefined—unless `x` and `y` are already defined elsewhere, in which case this will simply check that `p` is exactly equal to the value `(x,y)`. Use a boolean guard when you want to check some condition; use a pattern guard when you want to take a value apart or see what it looks like.

2.8.2 Function pattern-matching

As we have already seen, functions can be defined via multiple clauses and pattern-matching. In fact, any such definition simply desugars to one big case expression. For example, the `gcd` function shown below actually desugars to something like `gcd2`:

Listing 12: example/function-desugar.disco

```
gcd : N * N -> N
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)

gcd2 : N * N -> N
gcd2 = λp. {? a                when p is (a,0),
              gcd2(b, a mod b) when p is (a,b)
           ?}
```

2.8.3 Arithmetic patterns

Disco supports *arithmetic patterns*, in which arithmetic expressions involving numeric constants, variables, and arithmetic operations can be used as patterns. A few examples are shown below.

Listing 13: example/arith-pattern.disco

```
h : N -> N
h(0)    = 1              -- matches 0
h(2k+1) = h(k)           -- matches any natural of the form 2k+1 for a natural number␣
↪k
h(2k+2) = h(k+1) + h(k)  -- matches any natural of the form 2k+2

isHalf : Q -> Bool
isHalf(s) = {? true when s is _ / 2,  -- matches fractions with denominator 2
               false otherwise ?}
```

```
Disco> :load example/arith-pattern.disco
Loading arith-pattern.disco...
Loaded.
Disco> map(h, [0 .. 10])
[1, 1, 2, 1, 3, 2, 3, 1, 4, 3, 5]
Disco> isHalf(3/2)
true
Disco> isHalf(4/2)
false
Disco> isHalf(17)
false
Disco> isHalf(5/(-2))
true
```

In short, an arithmetic pattern can contain:

- variables
- natural number constants
- unary negation
- addition, subtraction, multiplication, and division

In most cases, an arithmetic pattern may contain at most one variable; for example, using `x + y` as an arithmetic pattern is an error, since the resulting values of `x` and `y` would be ambiguous. The one exception is when matching on an expression containing a division operator, in which case the two patterns are used to separately match on the numerator and denominator of the value being matched.

The behavior of arithmetic patterns depends on the type being matched. Generally speaking, matching will succeed if there is a unique value of the same type that can be assigned to the variable such that the pattern is equal to the value being matched. For example, when matching on natural numbers, the pattern `2k+3` will match only odd numbers greater than or equal to 3, since those are the only numbers which result from assigning a natural number value to `k`. Matching `2k+3` against an integer will match all odd integers; matching it against a rational number will always match.

An arithmetic pattern need not contain any variables, in which case it is the same as just matching on a particular constant.

At the moment, arithmetic patterns do not support exponentiation, though that could be a nice thing to add (but surely contains many pitfalls).

2.9 Lists

Disco defines a type of inductive, singly-linked lists, very similar to lists in Haskell.

2.9.1 Basic lists

All the elements of a list must be the same type, and the type of a list with elements of type `T` is written `List(T)`.

The basic syntax for constructing and pattern-matching on lists is almost exactly the same as in Haskell, with the one difference that the single colon (type of) and double colon (cons) have been switched from Haskell.

Listing 14: example/list.disco

```
emptyList : List(Bool)
emptyList = []

nums : List(N)
nums = [1, 3, 4, 6]

nums2 : List(N)
nums2 = 1 :: 3 :: 4 :: 6 :: []

  -- nums and nums2 are equal

nested : List(List(Q))
nested = [1, 5/2, -8] :: [[2, 4], [], [1/2]]

sum : List(N) -> N
sum []        = 0
sum (n :: ns) = n + sum ns
```

2.9.2 List comprehensions

Disco has list comprehensions which are also similar to Haskell's. A list comprehension is enclosed in square brackets, and consists of an expression, followed by a vertical bar, followed by zero or more *qualifiers*, `[ <expr> | <qual>* ]`.

A *qualifier* is one of:

- A *binding* qualifier of the form `x in <expr>`, where `x` is a variable and `<expr>` is any expression with a list type. `x` will take on each of the items of the list in turn.

- A *guard* qualifier, which is an expression with a boolean type. It acts to filter out any bindings which cause the expression to evaluate to `false`.

For example, `comp1` below is a (rather contrived) function on two lists which results in all possible sums of two *even* numbers taken from the lists which add to at least 50. `pythagTriples` is a list of all Pythagoren triples with all three components at most 100. (There are much more efficient ways to compute Pythagorean triples, but never mind.)

Listing 15: example/comprehension.disco

```
comp1 : List(N) -> List(N) -> List(N)
comp1 xs ys = [ x + y | x in xs, 2 divides x, y in ys, 2 divides y, x + y >= 50 ]

pythagTriples : List (N*N*N)
pythagTriples = [ (a,b,c)
  | a in [1 .. 20]
  , b in [1 .. 20]
  , c in [1 .. 20]
  , a^2 + b^2 == c^2
  ]
```

Note: The biggest difference between list comprehensions in disco and Haskell is that Haskell allows *pattern* bindings, *e.g.* `Just x <- xs`, which keep only elements from the list which match the pattern. At the moment, disco only allows variables on the left-hand side of a binding qualifier. There is no reason in principle disco can't support binding qualifiers with patterns, it just isn't a big priority and hasn't been implemented yet.

2.9.3 Polynomial sequences

Like Haskell, disco supports ellipsis notation in literal lists to denote omitted elements, although there are a few notable differences. One minor syntactic difference is that (just for fun) disco accepts two *or more* dots as an ellipsis; the number of dots makes no difference.

Listing 16: example/basic-ellipsis.disco

```
-- Counting numbers from 1 to 100
counting : List(N)
counting = [1 .. 100]

-- Even numbers from 2 to 100
evens : List(N)
evens = [2, 4 ..... 100]

-- [5, 4, 3, ... -3, -4, -5]
down : List(Z)
down = [5 .. -5]

-- 1 + 3 + 5 + 7 = 16
s : N
s = {? a+b+c+d  when [1, 3 .. 100] is (a::b::c::d::_) ?}

-- It doesn't always have to be integers
qs : List(Q)
qs = [2/3, 7/5 .. 10]
```

- `[a .. b]` denotes the list that starts with `a` and either counts up or down by ones (depending on whether `b` is greater than or less than `a`, respectively), continuing as long as the elements do not "exceed" `b` (the meaning of

"exceed" depends on whether the counting is going up or down).

- `[a, b .. c]` denotes the list whose first element is `a`, second element is `b`, and the difference between each element and the next is the difference `b - a`. The list continues as long as the elements do not "exceed" `c`, where "exceed" means either "greater than" or "less than", depending on whether `b - a` is positive or negative, respectively.

All the above is similar to Haskell, except that `[10 .. 1]` is the empty list in Haskell, and disco's rules about determining when the list stops are much less strange (the strangeness of Haskell's rules is occasioned by floating-point error, which of course disco does not have to deal with). Also, since disco is strict, it does not support infinite lists.

However, disco also generalizes things further by allowing notation of the form `[a, b, c .. d]` or `[a, b, c, d .. e]`, and so on. We have already seen that `[a, b .. c]` generates a linear progression of values; by analogy, `[a, b, c .. d]` generates a quadratic progression, `[a, b, c, d .. e]` a cubic, and so on. In general, when n values $a_0, a_1, \ldots, a_n$ are given before the ellipsis, disco finds the unique polynomial p of degree $n - 1$ such that $p(i) = a_i$, and uses it to generate additional terms of the list. (In practice, the disco interpreter does not actually find a polynomial, but uses the *method of finite differences*, just like Charles Babbage's Difference Engine.)

Listing 17: example/general-ellipsis.disco

```
-- Some triangular numbers
triangular : List(N)
triangular = [1, 3, 6 .. 100]

-- Some squares
squares : List(N)
squares = [1, 4, 9 .. 100]

-- Some cubes
cubes : List(N)
cubes = [1, 8, 27, 64 .. 1000]
```

The list continues until the first one which "exceeds" the ending value. The precise definition of "exceeds" is a bit trickier to state in general, but corresponds to the eventual behavior of the polynomial: the list stops as soon as elements become either larger than or smaller than the ending value, as the polynomial diverges to $+\infty$ or $-\infty$, respectively.

2.9.4 Multinomial coefficients

We already saw that the `choose` operator can be used to compute binomial coefficients. In fact, if the second operand to `choose` is a list instead of a natural number, it can be used to compute general multinomial coefficients as well. `n choose xs` is the number of ways to choose a sequence of sets whose sizes are given by the elements of `xs` from among a set of `n` items. If the sum of `xs` is equal to `n`, then this is given by `n!` divided by the product of the factorials of `xs`; if the sum of `xs` is greater than `n`, then `n choose xs` is zero; if the sum is less than `n`, it is as if another element were added to `xs` to make up the sum (representing the set of elements which are "not chosen"). In general, `n choose k = n choose [k,n-k] = n choose [k]`.

2.10 Polymorphism

Disco includes support for parametric polymorphism, with syntax similar to Haskell. For example, here is how we could write a polymorphic list map function (although this is actually built in to Disco; see the next section on containers).

Listing 18: example/poly.disco

```
maplist : (a -> b) -> List(a) -> List(b)
maplist _ [] = []
maplist f (a :: as) = f a :: (maplist f as)
```

Disco can also infer polymorphic types. For example:

```
Disco> :type \x,y. x
λx, y. x : a1 → a → a1
Disco> :load example/poly.disco
Loading poly.disco...
Loaded.
Disco> :type maplist (\x.x)
maplist (λx. x) : List a → List a
Disco> :type maplist (\x.x) [1, 2, 3]
maplist(λx. x)([1, 2, 3]) : List
```

However, although Disco has an internal notion of type qualifiers (like Haskell type classes), these will never show up in inferred types. For example:

```
Disco> :type \x,y. x + y
λx, y. x + y :  →  →
```

Internally, Disco is happy to use `\x,y. x + y` at any type which supports addition, but when forced to infer a concrete type for it, it simply picks a suitable monomorphic instantiation. However, the following example shows that it can in fact be used on, say, rational numbers:

```
Disco> :type (\x,y. x + y) (3/2) (-5)
(λx, y. x + y)(3 / 2)(-5) :
```

2.11 Type Definitions

The `type` keyword can be used to conveniently declare aliases for types. For example, consider the following function which takes a list of natural number triplets and returns the sum of all the triplets in the list:

Listing 19: example/tydefs.disco

```
sumTripletList : List (N * N * N) -> N
sumTripletList [] = 0
sumTripletList ((n1, n2, n3) :: rest) = (n1 + n2 + n3 + (sumTripletList rest))
```

```
Disco> sumTripletList [(1,2,3), (4,5,6)]
21
```

Let's write the following type definition:

```
type NatTriple = N * N * N
```

The type `NatTriple` is defined as a 3-tuple containing three values of type `N`. Note that in Disco, all type names must begin with a capital letter. Now we can rewrite our type declaration for `sumTripletList` as follows:

```
sumTripletList : List(NatTriple) -> N
```

2.11.1 Recursive type definitions

However, `type` definitions are in fact much more powerful. Disco has no special syntax for declaring algebraic data types as in Haskell, but unlike Haskell, `type` definitions in Disco can be recursive. Thus, we can build recursive algebraic data types directly. For example, we can define a type of binary trees with values of type `N` at nodes as follows:

Listing 20: example/tydefs.disco

```
type Tree = Unit + (N * Tree * Tree)
```

Here, we see that a `Tree` can either be a leaf, or a a triplet containing a natural number value of the root as the first element, and the left and right subtrees of type `Tree` as the second and third elements, respectively.

Given this definition of `Tree`, here is how we would write a function which takes a `Tree` and returns the sum of all its node values.

Listing 21: example/tydefs.disco

```
sumTree : Tree -> N
sumTree (left _) = 0
sumTree (right (n, l, r)) = n + sumTree(l) + sumTree(r)
```

2.11.2 Parameterized type definitions

Type definitions can also be parameterized. For example, we can make the `Tree` type polymorphic:

Listing 22: example/tydefs-poly.disco

```
import list

type Tree(a) = Unit + (a * Tree(a) * Tree(a))

treeFold : b * (a * b * b -> b) * Tree(a) -> b
treeFold (z, f, left(unit)) = z
treeFold (z, f, right (a, l, r)) = f(a, treeFold(z,f,l), treeFold(z,f,r))

sumTree : Tree(Nat) -> Nat
sumTree(t) = treeFold(0, \(a,l,r). a+l+r, t)

flattenTree : Tree(a) -> List(a)
flattenTree(t) = treeFold([], \(a,l,r). append(l, append([a], r)), t)
```

```
Disco> :load example/tydefs-poly.disco
Disco> leaf = left(unit)
Disco> t = right (1, right (3, leaf, leaf), right (5, leaf, leaf)) : Tree N
Disco> sumTree(t)
9
Disco> flattenTree(t)
[3, 1, 5]
```

2.11.3 Cyclic type definitions

There is only one restriction on recursive `type` definitions, namely, they are not allowed to be cyclic, *i.e.* unguardedly recursive. A `type` definition is cyclic if the following two conditions hold:

1. Repeated expansions of the `type` definition yield solely `type` definitions.
2. The same `type` definition is encountered twice during repeated expansion.

For example:

```
-- Foo is not allowed because it expands to itself.
type Foo = Foo
-- Bar is not allowed: it expands to Baz which expands to Bar.
type Bar = Baz
type Baz = Bar

-- Pair is OK (though rather useless) because it expands to a
-- top-level type former (product) which is not a type definition.
type Pair = Pair * Pair
```

2.12 Containers

Coming soon!

Topics to cover:

- warning: under construction
- **lists, bags, and sets**
  - literal syntax
  - conversion functions
- **built-in functions:**
  - `map`
  - `reduce`
  - `filter`
  - `join`
  - union, intersection, etc.
  - `merge`
  - `size`
- comprehensions

2.13 Properties

Each disco definition may have any number of associated *properties*, mathematical claims about the behavior of the definition which can be automatically verified by disco. Properties begin with `!!!`, must occur just before their associated definition, and may be arbitrarily interleaved with documentation lines beginning with `|||`.

2.13.1 Unit tests

The simplest kind of property is just an expression of type `Bool`, which essentially functions as a unit test. When loading a file, disco will check that all such properties evaluate to `true`, and present an error message if any do not.

Listing 23: example/unit-test.disco

```
!!! gcd(7,6)   == 1
!!! gcd(12,18) == 6
!!! gcd(0,0)   == 0

gcd : N * N -> N
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)
```

When we load this file, disco reports that it successfully ran the tests associated with `gcd`:

```
Disco> :load example/unit-test.disco
Loading example/unit-test.disco...
Running tests...
  gcd: OK
Loaded.
```

On the other hand, if we change the first property to `!!! gcd(7,6) = 2` and load the file again, we get an error:

```
Disco> :load example/unit-test.disco
Loading example/unit-test.disco...
Running tests...
  gcd:
  - Test result mismatch for: gcd (7, 6) = 2
    - Expected: 2
    - But got:  1
Loaded.
```

2.13.2 Quantified properties

More generally, properties can contain universally quantified variables. The syntax for a universally quantified property is as follows:

- the word `forall` (or the Unicode symbol ∀);
- one or more comma-separated *bindings*, each consisting of a variable name, a colon, and a type;
- a period;
- and an arbitrary expression, which should have type `Bool` and which may refer to the variables bound by the `forall`.

Such quantified properties have the obvious logical interpretation: they hold only if the given expression evaluates to `true` for all possible values of the quantified variables.

Listing 24: example/property.disco

```
!!!  x:Bool. neg (neg x) == x
neg : Bool -> Bool
neg x = not x

!!! p: N + N. plusIsoR (plusIso p) == p
plusIso : N + N -> N
plusIso (left n) = 2n
plusIso (right n) = 2n + 1
```

(continues on next page)

(continued from previous page)

```
!!! n:N. plusIso (plusIsoR n) == n
plusIsoR : N -> N + N
plusIsoR n =
  {? left  (n // 2)   if 2 divides n
   , right (n // 2)   otherwise
  ?}

!!! forall x:N, y:N, z:N.
      f(f(x,y), z) == f(x, f(y,z))

f : N*N -> N
f (x,y) = x + x*y + y
```

In the example above, the first three properties have a single quantified variable, and specify respectively that `neg` is self-inverse, and `plusIso` and `plusIsoR` are inverse. The last function has a property with multiple quantified variables, and specifies that `f` is associative. Notice that as in this last example, properties may extend onto multiple lines, as long as subsequent lines are indented. Only a single `!!!` should be used at the start of each property.

Such properties may be undecidable in general, so disco cannot automatically *prove* them. Instead, it searches for counterexamples. If the input space is finite and sufficiently small (as in the first example above, which quantifies over a single boolean), disco will enumerate all possible inputs and check each one; so in this special case, disco can actually prove the property by exhaustively checking all cases. Otherwise, disco randomly generates a certain number of inputs (*a la* `QuickCheck`) and checks that the property is satisfied for each. If a counterexample is found, the property certainly does not hold, and the counterexample can be printed. If no counterexample is found, the property "probably" holds.

For example, consider this function with a property claiming it is associative:

Listing 25: example/failing/property.disco

```
!!! forall x:N, y:N, z:N.
      f(f(x,y), z) = f(x, f(y,z))

f : N*N -> N
f (x,y) = x + 2*y
```

The function is not associative, however, and if we try to load this file disco quickly finds a counterexample:

```
Disco> :load example/failing/property.disco
Loading example/failing/property.disco...
Running tests...
  f:
  - Test result mismatch for:  x : , y : , z : . f (f (x, y), z) = f (x, f (y, z))
    - Expected: 5
    - But got:  3
    Counterexample:
      x = 1
      y = 0
      z = 1
Loaded.
```