Data types

In many programming languages, variables have types: e.g., `int32`, `char *`, or `dict`. The type of a variable tells us what kind of values we can assign to it: e.g, a variable of type `int32` can hold a signed integer that fits in 32 bits, using two's complement representation to handle negative numbers.

More importantly, a data type tells us an *interface*, a list of operations we can perform on values of this type. For example, there is an operation `+` that takes two `int32` values and produces another `int32` value (handling numerical overflow in a specified way).

The important thing about an interface is that it provides abstraction: it doesn't matter how an `int32` is represented under the hood, and in fact we rarely if ever have to think about this. We could even swap between two different implementations of `int32`; so long as they provide the same interface, our surrounding program shouldn't even notice.

Types as a formal system

A list of data types and their interfaces is a good example of a formal system: it tells us a mechanical way to generate and combine values of different types. If we follow the rules of the formal system, we're guaranteed to get values that make sense: e.g., we won't try to divide a `char *` by a `dict`. This guarantee applies no matter what specific implementations we use for the individual types.

What about an expression like `1/0`? Is this an exception to the above rule?

Example: the *set* data type

A set is an unordered collection of objects, without duplication. Each object is called an *element* of the set, written $o \in S$. Two sets are equal when they contain the same elements. A set X is a subset of a set Y, written $X \subseteq Y$, when all of the elements of X are also elements of Y.

For example, some useful sets are the integers $\mathbb{Z}$ or the real numbers $\mathbb{R}$. We have

$$3 \in \mathbb{Z},\ \pi \in \mathbb{R},\ \mathbb{Z} \subseteq \mathbb{R}$$

The above text is an informal description of an interface for a data type: it gives the name of the type and tells us what we can do with it. Here's a somewhat more formal description:

Given a type `E` (the type of objects that we can use as elements), we can define a new type called `set(E)` that supports the following operations:

- `set(E) make_set(E o1, E o2, ...)` creates a new set
- `bool contains(set(E) S, E o)` checks whether `o` is an element of `S`
- `bool subseteq(set(E) X, set(E) Y)` checks whether `X` is a subset of or equal to `Y`
- `bool equals(set(E) X, set(E) Y)` checks whether `X` contains the same elements as `Y`

We could add a few more operations to this interface if we wanted to: e.g., set unions and intersections, or modifying a set by adding or deleting elements. Depending on what options we pick, we get a *different* data type each time: the data type *is* the interface, so different interfaces mean different types.

Subtypes

As we saw above, it's often worth having multiple related interfaces: e.g., we might have one kind of set that supports adding and deleting elements, and another that does not. It would be a pain to have to think of these as completely separate types — we'd have to write a lot of code twice, violating the rule of "don't repeat yourself" and making our program harder to maintain.

For this reason, programming languages often support ways to create multiple related types and multiple related interfaces. The most common is probably subtyping by extension: given a type `T` with a particular interface, we can *extend* the interface by defining some new optionally-supported operations involving values of type `T`. The result is a new type `T'`, which is a *subtype* of `T`: only some values of type `T` will support the new operations, and so the possible values of type `T'` are a subset of the values of type `T`.

For example, we could extend our `set(E)` type above to support adding and deleting elements, to create a new type `extendable_set(E)` that supports the following additional operations:

- `extendable_set(E) insert(extendable_set(E) S, E o)` inserts a new element, if it is not already present (and does nothing otherwise)

- `extendable_set(E) delete(extendable_set(E) S, E o)` deletes an element, if it was present (and does nothing otherwise)

Because `extendable_set(E)` is a subtype of `set(E)`, we can take a value of type `extendable_set(E)` and interpret it as a value of type `set(E)` — that is, we can forget that we have the ability to add and delete elements. (This is called *type casting*.) But we can't go the other way: we can't magically force a value of type `set(E)` to become a value of type `extendable_set(E)`, since it might have been created in a part of the program that isn't even aware of the new insert and delete operations.

Set-builder notation

A really useful way to write new sets is with *set-builder notation*: we can either write a set by explicitly listing its elements,

$$\text{primary\_color} = \{\text{red}, \text{green}, \text{blue}\}$$

or by giving a recipe for constructing its elements

$$\text{even\_number} = \{2x \mid x \in \mathbb{Z}\}$$

The empty set $\emptyset$ is the set with no elements,

$$\emptyset = \{\}$$

The general form of a recipe is

$$S = \{\text{expression} \mid \text{property}_1, \text{property}_2, \ldots\}$$

The expression can contain variables like x, y, $\ldots$. The properties refer to the variables and tell us what values they can take: a legal value is one that satisfies all of the properties. S is then the set that contains all of the objects that we can get by picking legal values for all of the variables and substituting them into the expression.

A common shorthand is to write a simple logical property as part of the expression: for example, we can write the nonnegative integers as

$$\mathbb{N} = \{x \mid x \in \mathbb{Z}, x \geq 0\} = \{x \in \mathbb{Z} \mid x \geq 0\}$$

The comma in set-builder notation is shorthand for logical AND. So, these two expressions are equivalent:

$$\{x \mid x \in \mathbb{Z}, x \geq 0\} = \{x \mid x \in \mathbb{Z} \land x \geq 0\}$$

True or false: are these two sets the same? $\{2x \mid x \in \mathbb{Z}\} \overset{?}{=} \{x \in \mathbb{Z} \mid x \bmod 2 = 0\}$

Comprehensions

Set-builder notation is so useful that some programming languages implement a version of it. For example, Python has list comprehensions and set comprehensions: the Python code

```
{ x**2 for x in range(4) }
```

implements the set-builder notation

$$\{x^2 \mid x \in \{0,1,2,3\}\}$$

and therefore builds the set $\{0,1,4,9\}$. The clauses `for` and `if` in Python correspond to the logical properties in set-builder notation: `for` lists the range of a variable (like `for x in range(4)` above), and `if` filters by testing a property. So we can build the set of squares of even numbers between 0 and 10 with

```
{ x**2 for x in range(11) if x % 2 == 0 }
```

which returns

```
set([0, 64, 4, 16, 100, 36])
```

Just don't try to translate something like

$$\{2x \mid x \in \mathbb{Z}\}$$

directly into a Python comprehension...

Write a comprehension that uses Python's set-builder notation to find all the numbers between 1 and 7 that are not divisible by 2, 3, or 5. As a check, the output should be $\{1,7\}$.

A reminder: `x % 3` in Python gives the remainder when we divide `x` by `3`.

Set operations: union, intersection, difference, complement

After membership and subset testing, the next most basic operations on sets are union and intersection: in set-builder notation,

$$X \cap Y = \{x \mid x \in X, x \in Y\}$$

$$X \cup Y = \{x \mid x \in X \vee x \in Y\}$$

Another useful operation is set difference, everything that's in one set but not another:

$$X \setminus Y = \{x \mid x \in X, x \notin Y\}$$

(Note the use of $\notin$ as a shorthand for $\neg(x \in Y)$.)

Finally, sometimes we'll fix a universe U: a set that contains all possible objects we're considering as elements for other sets. (In our data type `set(E)` above, `E` is the universe.) Given a fixed universe, the set complement means everything that's not in a given set S:

$$S^C = \bar{S} = U \setminus S$$

Tuples and set products

Given two objects x and y, we write $\langle x, y\rangle$ for the ordered pair whose first element is x and second element is y. More generally, a tuple is a fixed-length list of values, like

$$\langle x, y, z, w\rangle$$

A tuple is a new data type, distinct from a set: e.g., in a tuple the order matters, while in a set it does not.

We can *nest* tuples, like

$$\langle\langle x, y\rangle, z, \langle w\rangle\rangle$$

and we can *flatten* them by removing internal pairs of angle brackets: if we flatten the above nested tuple, we get $\langle x, y, z, w\rangle$ again. In some situations it makes sense to flatten tuples implicitly, while in others it makes sense to distinguish nested tuples from their flattened counterparts.

Given two sets X and Y, the *set product* is the set of ordered pairs where the first element comes from X and the second comes from Y:

$$X \times Y = \{\langle x, y\rangle \mid x \in X, y \in Y\}$$

We use implicit flattening in this context, so that set product is associative:

$$X \times (Y \times Z) = (X \times Y) \times Z$$

and so we can write $X \times Y \times Z$ for both: the set of 3-tuples built from one element each of X, Y, and Z.

We'll write X^n for the set of n-tuples with elements in X: e,g.,

$$\langle 3, 1, 4, 1, 5 \rangle \in \mathbb{Z}^5$$

If X is the empty set and Y is the set $\{1, 2, 3\}$, what is $X \times Y$?

Combining types

Because a data type is based on a set of allowed values, we can use a lot of the above set operations to combine data types and make new ones. The difference is that when types are involved, we need to say how the *interfaces* combine as well.

An example is unions. In C, the expression

`union {int a, char *b}`

defines a type that we can interpret as the mathematical union between two simpler types: the type of values that are *either* integers or C strings. The interpretation is a bit subtle, though:

Where do we have to be careful when describing a C union type this way?

Whenever we use a value of the above C type, we have to know ahead of time whether we want to interpret it as an `int` or a `char *`. C is effectively treating both types as sets of bit strings and taking the union. By doing so, C is effectively violating our description of an interface: since a given bit-string could be interpreted either way, if we're not careful, we could assign an `int` to a variable, then read it as a `char *`, and get unexpected results. That is, C is providing a loophole that lets us use the `char *` interface on values of type `int`. There have been many heated arguments over whether this is a good idea. But certainly if we do it unintentionally, bad things can happen: e.g., we can get a program with a bad security vulnerability.

Tuples in programming languages

Many programming languages use tuples explicitly: e.g., in LISP (one of the oldest programming languages), a `cons` is a basic data type representing an ordered pair. The interface to a `cons` is two operations: `car` and `cdr` retrieve the first and second element of the pair, respectively. (Then we can use the appropriate interfaces to operate on these values individually.)

Or, in C, we can use `struct`s to represent tuples: the type

```
struct {int a, char *b}
```

represents an ordered pair (a length-2 tuple) of an integer and a C-style string. We can write the legal values for this type as a set product: $\mathbb{Z} \times \mathbb{S}$, where $\mathbb{S}$ stands for the set of C-style strings. (Well, almost: $\mathbb{Z}$ stands for the set of all integers, not just the ones that are representable in 32 bits.)

In databases like MySQL or Postgres, each database record is effectively a tuple. For example, a record in the student database could have fields for first name, last name, and mailing address. In both databases and C, we refer to the elements of a tuple by position names, while in LISP we refer to them by position numbers; these are different interfaces, but equivalently powerful.

Lots of languages use tuples implicitly as well. For example, when we define a function like

```
void main(int argc, char *argv[])
```

the argument list can be viewed as a tuple: in this case an ordered pair of an argument count and a C-style array of C-style strings.

Disjoint unions

Let's return to the C union type above. What we might expect, instead of C's default behavior, is that we know whether every value is intended as an integer or a string. This kind of combination of types is called a *tagged union* or a *disjoint union*: the same bit string is treated differently depending on whether it arrived as an `int` or a `char *`. Disjoint unions prevent us from (accidentally or on purpose) treating an object of one type as if it were an object of another type.

We can achieve the same effect in a programming language by requiring the types in a union to be disjoint. While C doesn't do this, some languages use the first few bits of a value to indicate its type. Or, we can achieve a similar effect in C by manually tagging the elements: e.g.,

```
struct {int tag, union {int a, char *b}}
```

It should be clear at this point that it's important to be explicit about which convention we're using: we have to pick a single formal system and stick with it if we want to get results that make sense. Either we do or we don't automatically remember what type of value we're currently storing, but if we think we do when we don't, bad stuff can happen.

Functions

One of the most interesting ways to combine types is to define a function: e.g.,

$$p(x, y) = 3x^2 + xy$$

The function p here is an object of a new type, which we can write as

$$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

This expression stands for the set of functions that take two real arguments and return a single real output.

In the above expression the new type is implicit: we infer it automatically based on the assumed types of the arguments.

Some programming languages are like this as well; e.g., Python doesn't require us to annotate function types, and will infer types automatically. (In fact, Python is quite aggressive about guessing types; this can be convenient but sometimes leads to unexpected behavior.) In other languages, like C, we have to annotate function definitions explicitly with input and output types: e.g.,

```
float p(float x, float y) { ... }
```

If we have a function $p \in X \to Y$, then for every value $x \in X$, there is a unique value $p(x) \in Y$. That is, functions are *complete* (defined for every input) and *single-valued* (never have ambiguous outputs).

An alternate notation for a function type $X \to Y$ is Y^X. This is by analogy to the set product notation Y^n: in Y^n we assign a unique value of type Y to every integer in $1 \ldots n$. Similarly, in Y^X we assign a unique value of type Y to every input value of type X.

Anonymous functions

When we talk about function types, it's convenient to use anonymous functions — that is, functions without explicit names. In C, all functions have to have names:

```
float square(float x) { return x**2; }
```

But sometimes the name is arbitrary, so it's useful to be able to skip it. For example, it's much shorter just to talk about a function x^2 instead of having to write out the above definition of `square`.

To define an anonymous function, we can use *lambda notation*: the expression

$$\lambda x, y.\, 3x^2 + xy$$

tells us the argument list and the definition for a function. With lambda notation it's common to leave types implicit: if we assume real inputs, then the above function has type $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. But we can explicitly annotate the types if desired, either just the input types:

$$\lambda x : \mathbb{R}, y : \mathbb{R}.\, 3x^2 + xy$$

(in which case we can infer the output type from the definitions of addition, multiplication, and exponentiation) or the output type as well:

$$\lambda x : \mathbb{R}, y : \mathbb{R}.\, (3x^2 + xy) : \mathbb{R}$$

In general, the syntax is

$$\lambda \text{ argument\_list. expression}$$

where argument_list is a comma-separated list of input variables, optionally with types, and expression tells us how to compute the function.

Many programming languages implement anonymous functions. For example, in LISP, we can write the above function as

```
(lambda (x y) (+ (* x x) (* 3 x y)))
```

while in Python we can write

```
lambda x, y: x*x + 3*x*y
```