1 `int`

1.1 Explained

Here, we just encode values as `int`s regardless of their units.

1.2 Example

```
signature UNITS = sig
    type miles = int
    type km = int
end

(* "Good" code *)
val d1 : miles = 12
val d2 : km = 15

(* Intern code *)
val d3 = d1 + d2
```

Pros:

- Easy to use?

Cons:

- Compiler won't check anything
  - If you forget an annotation, you have no idea what the units are
  - If you make a mistake, it may go unnoticed

Of course this would never happen in a real codebase...except that time it did and it cost NASA $327 million.

2 ADTs

2.1 Explanation

Here, we use an ADT to represent distance, representing different units as different variants of the `datatype`. This is a pretty standard technique in SML and similar languages.

2.2 Example

```
signature UNITS = sig
    datatype distance = Miles of int | Km of int

    val miles_to_km : distance -> distance
    val km_to_miles : distance -> distance

    (* Arguments must be in same units *)
    val op+ : distance * distance -> distance
end

(* Good code *)
val d1 = Miles 12
val d2 = Km 15

(* Intern code - still compiles but will fail at runtime *)
val d3 = km_to_miles d1
val d4 = d2 + d3
```

Pros:

- Units are checked

Cons:

- Units are checked *at runtime* - no way to statically check or constrain units.
- Must pay runtime cost for checks even if you never make any mistakes

3 Phantom Types

3.1 Explained

A *phantom type* refers to a type parameter which is not used by values of that type. For example, consider the following:

```
type 'a t = int
```

Even though `t` has a type parameter, values of type `'a t` have nothing to do with `'a` - they're just `int`s. Therefore, we refer to `t` as a phantom type.

Phantom types are useful for making the typechecker perform compile-time enforcement of various things without doing anything at runtime.

In the following example, we use phantom types to encode the units of `distance` values in their types.

3.2 Example

```
signature UNITS = sig
    type miles
    type km

    type 'u distance

    val miles : int -> miles distance
    val km : int -> km distance

    val miles_to_km : miles distance -> km distance
    val km_to_miles : km distance -> miles distance

    val op+ : 'u distance * 'u distance -> 'u distance
end

(* Good code *)
val d1 = miles 12
val d2 = km 15

(* Formerly intern code - now a type error *)
```

```
val _ = d1 + d2
val _ = km_to_miles d1

(* SML doesn’t let us case on what type ’u is *)
fun to_km (d : ’u distance) : km distance = ???
```

Pros:

- Unit safety is statically enforced

Cons:

- Can’t case on units
- Can’t write generic conversion functions - we would need to write a function for every possible pair of units
- Can write nonsensical types like `string distance`

4 GADTS

***Note**: Unlike all the previous examples, GADTs are not valid SML. There are other, similar-ish, languages with GADTs such as OCaml and Haskell in which a similar example would work just fine.*

4.1 Explained

Generalized algebraic data types, or GADTs, add additional functionality to the ADTs we know and love. Consider the following code:

```
datatype 'a option =
    NONE
  | SOME of 'a
```

If we write out the types of the constructors, we can see that

```
NONE : 'a option
SOME : 'a -> 'a option
```

Knowing this, we could change the syntax of `datatype` declarations to define the types of constructors instead of the arguments they take in. Such a declaration would look like the following:

```
datatype 'a option =
    NONE : 'a option
  | SOME : 'a -> 'a option
```

Notice how every constructor returns an `'a option`. This is required for regular ADTs. What GADTs do (and what makes them "generalized") is that they let you write constructors returning a type more specific than `'a option`. For example, we could write the following:

```
datatype 'a option =
    NONE : 'a option
  | FOO : unit option
  | SOME : 'a -> 'a option
```

This has several consequences when casing on a value of type `'a option`. For example, consider the following:

```
fun no_unit_club (x: bool option) : unit =
    case x of
        NONE => ()
      | SOME _ => ()
```

Note that this code doesn't consider the `FOO` case, and yet *the compiler won't complain* - not even a nonexhaustive match warning. That's because the compiler knows that a `bool option` can't have the form `FOO`, since `FOO` is a `unit option`.

Another surprising consequence is that values can have different types in different case arms. Consider the following:

```
datatype 'a tagged =
    Bool : bool tagged
  | Int : int tagged

fun extract (x : 'a tagged) : 'a =
    case x of
        Bool b => b     (* x : bool tagged *)
      | Int i => i      (* x : int tagged *)
```

Note that depending on which case arm you look at, `x` has a different type! This is very different from what we're used to. Additionally, consider the following:

```
val _ = extract (Bool true)
```

In this example, the argument to `extract` has type `bool tagged`. Because of this, the compiler could, without any complex analysis, statically determine that only the `Bool` case can be taken and elide the branch[1].

In the following example, we modify the ADT code to use GADTs. In a way, this is a combination of the ADT and phantom type approaches - we use a `datatype` declaration but tag `distance`s with a type representing

[1]Such an optimization certainly *could* be done without the type information, but optimizers can be unpredictable and the type information makes the optimization much simpler.

their units. Unlike either previous approach, however, the constructors are associated with their units at the type level.

4.2 Example

```
signature UNITS = sig
    type miles
    type km

    datatype 'u distance =
        Miles : int -> miles distance
      | Km : int -> km distance

    val miles_to_km : miles distance -> km distance
    val km_to_miles : km distance -> miles distance

    val op+ : 'u distance * 'u distance -> 'u distance
end

(* Good code *)
val d1 = miles 12
val d2 = km 15

(* Formerly intern code - now a type error *)
val _ = d1 + d2
val _ = km_to_miles d1

fun to_km (d : 'u distance) : km distance =
    case d of
        Miles _ => miles_to_km d    (* d: miles distance *)
      | Km _ => d                   (* d: km distance    *)

(* New intern code *)
fun to_miles (d : 'u distance) : unit =
    case d of
        Miles _ => d
      | Km _ => (print "I hate metric"; Miles 0)
```

Pros:

- Unit safety is statically enforced
- We can case on units

Cons:

- Type inference becomes more complicated
- Breaks the intuitive notion of parametricity - polymorphic functions may behave differently when applied to different types