```
{-
These lines declare that `A` and `B` are principles, and the type checker
(not yet implemented) will make sure that all principles mentioned in types
are (at least) well-scoped, and possibly also check some security relationships
as well, e.g., does A trust B, and to what extent.

These lines are ignored by the interpreter.
-}
principal A
principal B

{-
`cmp` takes a secret input from both `A` and `B` and builds a circuit which
compares them, returning a boolean.

The input variable `xy` has type `ℤ[64]{isec:A,B}`, which reads as “a 64-bit
integer as independent (i.e., not necessarily equal) secrets for each party `A`
and `B`.” To access `A`'s value, we write `xy.A`, and likewise for `B` and
`xy.B`. `xy.A` is a secret value, and can be embedded inside of a circuit with
`~xy.A`. All parties share this circuit, and `A`'s value inside `~xy` will be
provided as an input to the circuit when executed using a protocol (e.g. yao).

The access mode `isec` is distinguished from `ssec`, which reads as “shared
secret”, in that `ssec` values have the added invariant that all parties have a
local copy of the exact same value. E.g., when a result from mpc is revealed to
some subset of parties. `ssec` values can be used to construct new circuits,
whereas `isec` values cannot, or else we would have two parties evaluating
different circuits, which would be bad.

The return type `𝔹{ccir:A,B}` reads as “a circuit comprised of comparison
operations on numeric types with embedded secret inputs from `A` and `B`”.
The fact that a comparison operator is embedded in the circuit computation is
tracked coarsly at the type level. E.g., it may not be possible to execute a
`ccir` using bgw, and we want to throw a runtime error (and type error) if the
programmer tries to do it.

The first two steps of the function embed secret inputs from `A` and `B` into
single-node circuits—they are degenerate circuits with just the embedded secret
value. The type of these circuits is `ncir`, which says “no operations”. The
last step is a normal comparison operator which is overloaded syntactically to
operate over circuit values, giving a circuit result type `ccir` which tracks
the comparison operation.
-}
def cmp : ℤ[64]{isec:A,B} → 𝔹{ccir:A,B}
def cmp = λ xy →
  let x : ℤ[64]{ncir:A}
  let x = ~xy.A
  let y : ℤ[64]{ncir:B}
  let y = ~xy.B
  let r : 𝔹{ccir:B}
  let r = x ≤ y
  in r

{-
We have moved away from a monadic discipline for tracking effects, and are now
using a more traditional effect system, e.g., an effectful computation is a
function of a dummy argument `•` with an effect annotation over the arrow. The
effect in this function is `{inp:A,B;rev:}` which means “inputs are read from
parties `A` and `B`, but nothing is revealed to any part”. The thinking for now
is that it may be advantageous to track security-critical events in effect
types, at the very least for auditing. If I am using a library function or some
old code, what I really want to know from a security point of view is whether
or not (1) secret inputs are read from disk, (2) any mpc protocols are
executed, and (3) what parties learned the results of mpc. In particular, if a
function has no effect, we know none of these things have happened inside the
function, and we may decide it's not worth our time to audit (from a security
perspective). I realize this isn't the whole story, e.g., we may be crucially
depending on the functionality of some library function to have the correct
ideal functionality, but the idea is here, and we can keep it around and
discuss or drop it.

The return type is `𝔹{yshare:A,B}` which reads as “a boolean which is encrypted
and recoverable by `A` and `B` by combining encrypted values (e.g., secret
shares), and readily embeddible inside a future circuit which is later executed
using yao”. Basically this function has decided to execute the circuit `cmp`
using yao, and what comes back is like a secret share, and it can either be
revealed, or embedded inside a future yao computation.

Although this doesn't occur in this example file, suppose you took two values
`a,b : 𝔹{yshare:A,B}` and xor'd them like so `a ⊕ b`. The resulting type would
be `𝔹{bcir:yshare:A,B}`, that is, a circuit with boolean operations and
embedded yao encrypted values. Maybe we should just call this `𝔹{ycir:A,B}`,
because the only protocol it makes sense to run it with is yao.

The first step of `cmp-mpc` enters parallel mode for both `A` and `B` by the
syntax `{A,B}`. The read operation will see a local file "e1-input.txt" for
each party `A` and `B`. In our simulator, these live in `examples-data/A` and
`examples-data/B`. The result is an `isec:A,B` which again means “independent
secrets for both A and B”.

The final step of `cmp-mpc` executes the circuit constructed by `cmp xy` using
`yao` between parties `A` and `B`. We could have written, e.g., `mpc{yao:C,D}`
which would have resulted in (1) A and B send secret-shares of their inputs to
C and D, (2) C and D execute yao protocol, and (3) C and D end up with
(secret-share) encrypted results which can later be revealed.
-}
def cmp-mpc : 𝟙 →{inp:A,B;rev:} 𝔹{yshare:A,B}
def cmp-mpc = λ • →
  let xy : ℤ[64]{isec:A,B}
  let xy = {A,B} read[ℤ[64]] "e1-input.txt"
  let r : 𝔹{yshare:A,B}
  let r = mpc{yao:A,B} cmp xy
  in r

{-
`cmp-mpc-rev` run `cmp-mpc` and then reveals the result to each other. Here the
`reveal` expression takes as input `𝔹{yshare:A,B}` and returns as output
`𝔹{ssec:A,B}`. Just as with `mpc`, you could write `reveal{C,D}` and the
meaning of that program is that `A` and `B` send their shares to both `C` and
`D`, who then recombine them to learn the result.
-}

def cmp-mpc-rev : 𝟙 →{inp:A,B;rev:A,B} 𝔹{ssec:A,B}
def cmp-mpc-rev = λ • →
  let r : 𝔹{yshare:A,B}
  let r = cmp-mpc •
  let p : 𝔹{ssec:A,B}
  let p = reveal{A,B} r
  in p

{-
`main` just runs `cmp-mpc-rev` to completion. Running this example shows the
result of `main`
-}
def main : 𝔹{ssec:A,B}
def main = cmp-mpc-rev •
```