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

These lines are ignored by the interpreter.

In this example, A and B have secret inputs. C and D will recieve
shares of A and B's secret inputs in order to compute over them
through an MPC protocol. E and F will then recieve output shares
from C and D in order to reconstruct the answer, which is only
learned by E and F (not A, B, C or D).
-}
-- input parties
principal A
principal B
-- compute parties
principal C
principal D
-- output parties
principal E
principal F

{-
`cmp` takes a secret input from both `A` and `B` and builds a yao
boolean
“secret share” which is (conceptually) immediately sent to (and
shared by) C and D. Specifying yao at this point in the program (as
opposed to when the final results is “revealed”) allows for (e.g.)
EMP-style on-they-fly evaluation of a protocol as an optimization.

The input variable `xy` has type `ℤ{isec:A,B}`, which reads as “two
integers, one for each party `A` and `B`, as independent (i.e., not
necessarily equal) secrets.” To access A's value, we write `xy.A`,
and likewise for B and `xy.B`. `xy.A` is a secret value, and is (yao)
“secret-shared” (i.e., encrypted and embeddable inside of a circuit)
to C and D by `share{yao:C,D} xy.A`. C and D now hold shares of A's
secret which will be provided as an input to the next circuit (in
this case, a yao garbled circuit with a single comparison) when
executed.

The return type `𝔹{yao:C,D}` reads as “A boolean value embedded in a
yao secret share that is shared among C and D.” The type checker will
check for valid operations to be used in a mpc computations. In this
example, comparison (≤) is valid because it is supported by the yao
prtocol. E.g., it may not be possible to execute a circuit that uses
comparisons using bgw, so the type checker would reject a program
that tried to do so. In this example, we have already commited to yao
but that may not always be the case. Regarding the `{inp:A,B}` part
of the function type, this annotation specifies from which parties
were secret inputs secret-shared and computed on in some protocol.
This enables checking security relationtionships between principles
and flagging violations of the trust model. The interpreter ignores
input effect annotations such as this.

The first two steps of `cmp` embed secret inputs from A and B into
yao “secret-shares”. The last step is a normal comparison operator
```

which is overloaded syntactically to operate over secret share values by executing the protocol over the secret-share. Since we are using yao, we can use the `≤` operator unlike in say, bgw, where comparison like this isn't possible. In the future it will be possible to pick whether or not protocol execution operaties on-the-fly with small circuits (e.g., in EMP), or as one large circuit in a single protocol. So conceptually `𝔹{yao:C,D}` is a secret share, and `x ≤ y` executes a circuit of size 1, but in practice `𝔹{yao:C,D}` could be a circuit constucted by each party, `x ≤ y` builds a larger circuit out of smaller ones, and execution of the protocol is delayed until some later point.

```
-}
def cmp : ℤ{isec:A,B} →{inp:A,B} 𝔹{yao:C,D}
def cmp = λ xy →
  let x : ℤ{yao:C,D}
  let x = share{yao:C,D} xy.A
  let y : ℤ{yao:A,B}
  let y = share{yao:C,D} xy.B
  let r : 𝔹{yao:C,D}
  let r = x ≤ y
  in r
{-
```

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.

```
-}
def cmp-mpc : 𝟙 →{inp:A,B;rev:E,F} 𝔹{ssec:E,F}
def cmp-mpc = λ • →
  let xy : ℤ{isec:A,B}
  let xy = {par:A,B} read ℤ "e1-input.txt"
  let r : 𝔹{yao:C,D}
  let r = cmp xy
  let o : 𝔹{ssec:E,F}
  let o = reveal{E,F} r
  in o

def one-liner : 𝟙 →{inp:A,B;rev:E,F} 𝔹{ssec:E,F}
def one-liner = λ • →
  let xy = {par:A,B} read ℤ "e1-input.txt"
  in reveal{E,F} (share{yao:C,D} xy.A) ≤ (share{yao:C,D} xy.B)

def main : 𝔹{ssec:A,B} × 𝔹{ssec:A,B}
def main = cmp-mpc • , one-liner •
```