Boolean Example Program

A Pict program encoding booleans as π calculus processes:

```pict

type Boolean = ^[^[] ^[]]

def tt[b:Boolean] = b?[t _] = t![[]]
and ff[b:Boolean] = b?[_ f] = f![[]]

def test[b:Boolean] =
    (new t:^[] new f:^[]
        (b![t f]
            |t?[] = print!"True"
            |f?[] = print!"False"))

new b:Boolean
run (ff![b] | test![b])
```

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 8.3, 8.4 – p. 2/34
Type Refinement and Subsumption

Types can be refined to subtypes enabling only certain operations:

- \(^T\) is a channel type.
- \(!T\) is an output channel type.
- \(?T\) is an input channel type.
- \(/T\) is a responsive output channel type.

```plaintext
type Boolean = ^[^[] ^[]]
type ClientBoolean = ![^[] ^[]]
type ServerBoolean = ?[^[] ^[]]
```

- Boolean \(<\) ClientBoolean
- Boolean \(<\) ServerBoolean

Boolean is a subtype of both ClientBoolean and ServerBoolean.
Type Refinement and Subsumption

type Boolean = ^[^[] ^[]]

type ClientBoolean = ![^[] ^[]]

type ServerBoolean = ?[^[] ^[]]

def tt[b:ServerBoolean] = b?[t _] = t![[]

and ff[b:ServerBoolean] = b?[_ f] = f![[]

def test[b:ClientBoolean] =

    (new t:^[] new f:^[]

        (b![t f]

            |t?[] = print!"True"

            |f?[] = print!"False")

    )

new b:Boolean

run (ff![b] | test![b])

CSCL.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 8.3, 8.4 – p. 4/34
Type Refinement and Subsumption

In general,

- \(^T < ?T.
- \(^T < !T.

Top is the super-type of every other type in Pict: “don’t care”-type.
Responsive Output Channels

The type */T is used for channels that communicate values of type \( T \), except that these channels have been created using a process definition (\( \text{def} \)). This ensures that:

- it is the same receiver process that always responds to inputs over the channel.
- it is **always** available to receive values.

For example, + is a built-in channel that receives a tuple with two numbers and returns the addition in the third argument.

- Its type is therefore */[Int Int /Int].
- We could use it as follows:

```
run (def r x:Int = printi!x
    +![2 3 r])
```
Responsive Output Channels

A channel created by `def` has the type `/T` (responsive channel), e.g.:

```plaintext
new x: ^[/Bool]
def d b:Bool = if b then print!"True"
    else print!"False"
run ( x![d]
    | x?[a] = a!false )
```

`/T < !T` holds, e.g.:

```plaintext
new y: ^[!Bool]
def d b:Bool = if b then print!"True"
    else print!"False"
run ( y![d]
    | y?[a] = a!false )
```
Responsive Channels

Many Pict standard libraries use responsive channels, e.g.:

```plaintext
pr /[String /[]]
```

`pr` is a responsive channel expecting a `String` and a responsive channel to signal completion.

`/[]` is a very common type, its name is `Sig`. (type `Sig = /[]`)

```plaintext
pr /[String Sig]
```

```plaintext
def d [] = print!"Done"
run pr!"pr..." d
```

Coercing ordinary into responsive channels:

```plaintext
new c : ^[]
run ( pr!"pr..." (rchan c)]
     | c?[] = print!"Done")
```

**Exercise:** Use `+` with an ordinary channel.
Reference Cell in Pict

Using the reference cell with ordinary channels:

```plaintext
new contents:^Int
run contents!0
def get [res:!Int]
    = contents?v = ( contents!v | res!v )
def set [v:Int c:Sig]
    = contents?_ = ( contents!v | c![[]] )

new done:^[]
new res:^Int
run (set![5 (rchan done)]
    | done?[] =
        ( get![(rchan res)]
        | res?i = printi!i))
```
Reference Cell in Pict (2)

Using the reference cell with responsive channels:

```plaintext
new contents:^Int
run contents!0
def get [res:!Int]
    = contents?v = ( contents!v | res!v )
def set [v:Int c:Sig]
    = contents?_ = ( contents!v | c![] )

def res i:Int = printi!i
def done [] = get![res]
run set![5 done]
```
Functional Programming

Pict offers convenient derived forms that enable viewing processes as functions, and therefore allow to program in Pict as if it were a functional programming language.

The convention in Pict is that a process definition that takes a tuple whose last parameter is used for output:

\[
\text{f} \quad \text{/[T}_1 \ldots \text{T}_n \quad \text{/T}]
\]

can be thought of as a function, and syntactic sugar is provided so that it can be written as if it were a function definition:

\[
\text{def f(a}_1\text{:T}_1\ldots\text{a}_n\text{:T}_n\text{):T } = \text{v} \quad \triangleq \quad \text{def f [a}_1\text{:T}_1\ldots\text{a}_n\text{:T}_n \quad \text{r:/T} ] = \text{r!v}
\]
Using and defining functions

Recall the + built-in channel of type /[Int Int /Int]:

```
run (def r x:Int = printi!x
     +![2 3 r])
```

Since the process reading on the channel + follows the convention of functions, we can make the channel r implicit and write (+ 2 3) as if it were a value:

```
run printi!(+ 2 3)
```

Similarly, we can define a double process as:

```
def double[x:Int r:/Int] = +![x x r]
```

or we can use the equivalent *functional* style:

```
def double(x:Int):Int = (+ x x)
run printi!(double 5)
```
Anonymous abstractions

Furthermore, there is syntactic support for anonymous abstractions:

\[ a \triangleq (\text{def } x \ a \ x) \]

For example,

```scala
def double(x:Int):Int = (+ x x)
def applyTwice (f:/\[Int /Int\] x:Int):Int = (f (f x))
run printi!(applyTwice double 3)
```

can also be written as:

```scala
def applyTwice (f:/\[Int /Int\] x:Int):Int = (f (f x))
run printi!(applyTwice \(x) = (+ x x) 3)
```

or as:

```scala
run printi!(\(f:/\[Int /Int\] x:Int):Int = (f (f x))
  \(x) = (+ x x) 3)
```
For loop example

A loop example using an anonymous abstraction and regular channels:

```r
def for [i:Int j:Int p:/[Int Sig] c:/[]] =
  if (<= i j) then
    (new d:^[]
      ( p![i (rchan d)]
        | d?[] = for![(+ i 1) j p c])
    )
  else c![[]

new d:^[]
run ( for![1 5
  \[x:Int c:Sig] = (printi!x | c![[])
  (rchan d)]
  | d?[] = print!"Done" )
```
For loop example (2)

A loop example using an anonymous abstraction and process definitions (responsive channels):

```plaintext
def for [i:Int j:Int p:/[Int Sig] c:][/[]] =
    if (<= i j) then
        (def d [] = for![(+ i 1) j p c]
        p![i d])
    else c![]

def d [] = print!"Done"

run for![(1 5)
    \[x:Int c:Sig] = (printi!x | c[])
    d]
```
Value Declarations and Sequencing

One common programming pattern is to ensure that an operation has ended before starting another one. Pict offers blocking value declarations as well as signaling that enable processes to synchronize and execute sequentially.

For example, suppose two processes write “hello” and “world” respectively to standard output:

```
run ( print!"hello"
    | print!"world")
```

Since \((e_1 | e_2) \equiv (e_2 | e_1)\), this program could print these words in any order: helloworld or worldhello.
Value Declarations and Sequencing

The most obvious way to synchronize between two processes is to communicate a value over a shared channel. For example:

```
run (new c:^[]
    ( pr!["hello" (rchan c)]
    | c?[] = print!"world")
```

In this example, we create a channel c which is used by the first process to signal completion to the second process. This ensures that “hello” is printed before “world”.

We have used a primitive called pr that prints to the standard output stream (like print,) but also receives a responsive channel where it can send a completion signal to indicate that it has finished its work.

The following code also achieves the same effect:

```
run (def c[] = print!"world"
    pr!["hello" c])
```
Value Declarations and Sequencing

This “continuation-passing style” communication is so common that there is syntactic sugar for it:

\[(\text{val } p = v \ e) \triangleq \text{(new c } (c?p = e \mid c!v))\]

Notice that in the translation of a \((\text{val } p = v \ e)\) expression, the body \(e\) is guarded by an input over the continuation channel. That means that the behavior of the expression is to \textit{block} until a value \(v\) has been computed and pattern matched against \(p\).

Using this syntactic sugar, the example could be written as follows:

```plaintext
run(val [] = (pr "hello")
    print!"world"
)
```
Value Declarations and Sequencing

The pattern “invoke an operation, wait for a signal, and continue” is so common that Pict offers syntactic sugar for it:

\[ v; \mid val[\ ] = v \]

Or, it can be written as follows:

```plaintext
run((pr "hello");
    print!"world"
)
```

The `prNL` function prints a string and moves to a new line. Its type is `/[String Sig]`. 
For loop example (3)

A loop example using an anonymous abstraction, functional notation, and sequencing:

```scala
def for(i: Int j: Int p: Int Sig): [] =
  if (i <= j) then
    ((p, i);
     (for (+ i 1) j p))
  else []

run ((for 1 5

  \[x: Int c: Sig = (printi!x | c![]));

  (prNL "Done"));

  ()))
```

Because output is asynchronous, we are not guaranteed to send the signal on `c` after printing.
For loop example (4)

```scala
def for(i:Int j:Int p:/[Int Sig]):[] =
  if (<= i j) then
    ((p i);
     (for (+ i 1) j p))
  else []

run ((for 1 5
   \[x:Int c:Sig] = prNL![(int.toString x) c]);
     (prNL "Done"));
   ())
```

Here, we use the `prNL` process to ensure that a new iteration only occurs after the printing has occurred.

The `int.toString` function takes an integer argument and returns its string representation. Its type is `/[Int /String]`. 
For loop example (5)

A loop example using an anonymous functional abstraction:

```scala
def for(i:Int j:Int p:/(Int Sig)):[Int] = 
  if (<= i j) then 
    ((p i);
     (for (+ i 1) j p))
  else []

run ((for 1 5 
   \(x:Int)[] = (prNL (int.toString x)));
   (prNL "Done");
   ()
```

We use `prNL` as a function in the anonymous abstraction. Channel `c` became implicit.
Using the reference cell with functional notation and sequencing:

```plaintext
new contents:^Int
run contents!0
def get [res:!Int]
    = contents?v = ( contents!v | res!v )
def set [v:Int c:Sig]
    = contents?_ = ( contents!v | c![[]] )
run ((set 5);
    (prNL (int.toString (get)));
    ()
)
```
The reference cell that returns a tuple with the get and set servers.

```plaintext
def refInt [res:(/[[/Int] /[Int Sig]])] =
    (new contents:^Int
     run contents!0
     def get [res:!Int]
         = contents?v = ( contents!v | res!v )
     def set [v:Int c:Sig]
         = contents?_ = ( contents!v | c![] )
     res![get set]
    )
```
Therefore, a client of the reference cell can be coded as follows:

```haskell
val [g1 s1] = (refInt)
val [g2 s2] = (refInt)

run ((s2 5);
    (prNL (int.toString (g1)));
    (prNL (int.toString (g2)));
    ()
)
```
Object-based programming

An object has:

- internal (encapsulated) state
- a set of operations on the state

For example, our reference cell has:

- a contents channel with the cell’s value
- a get and a set operation to observe/modify the cell’s value
Reference Cell Object

We can define the type and return a record with `get` and `set`:

type RefInt = [
    get = /[/Int]
    set = /[[Int Sig]
]

def refInt () : RefInt =
( new contents:^Int
    run contents!0
    [get = \[res:/Int] =
        contents?v = (contents!v | res?v)
    set = \[v:Int c:Sig] =
        contents?_ = (contents!v | c![])
  ])

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Sections 8.3, 8.4 – p. 27/34
Reference Cell Object

The new definition can be used as follows:

```
run ( val ref1 = (refInt)
    val ref2 = (refInt)
    (ref2.set 5);
    (prNL (int.toString (ref1.get)));
    (prNL (int.toString (ref2.get)));
    ()
  )
```

Notice that using a record to return the different “operation servers” on an encapsulated state starts to resemble object-based programming.
Lists

Standard functional programming primitives for lists: cons, car, cdr.

```plaintext
import "Std/List"
val l = (cons 6 (cons 7 ( cons 8 nil)))
run printi!(car (cdr l))
```

Prints 7.
Folding

Syntactic support for folding:

\[(\text{cons} > 6 \ 7 \ 8 \ \text{nil})\]

≡

\[(\text{cons} 6 \ (\text{cons} 7 \ (\text{cons} 8 \ \text{nil})))\]

More generally:

\[(f > a_1 \ a_2 \ldots \ a_n \ a)\]

≡

\[(f \ a_1 \ (f \ a_2 \ldots \ (f \ a_n \ a)\ldots))\]

Also right folding:

\[(f < a \ a_1 \ a_2 \ldots \ a_n)\]

≡

\[(f\ldots(f \ (f \ a \ a_1) \ a_2) \ldots \ a_n)\]
Polymorphism

```scheme
def print2nd[#X l:(List X) p:/[X /String]] =
  if (null l) then print!"Null list"
else if (null (cdr l)) then print!"Null tail"
else print!(p (car (cdr l)))
```

The # indicates it is a type parameter, e.g.:

```scheme
run print2nd! [#Int (cons > 6 7 8 nil)
  int.toString]
```

prints 7.

```scheme
run print2nd! [#String (cons > "one" "two" nil)
  \(s:String) = s]
```

prints two.
Generic Reference Cell Object

We can define a generic type for reference cells:

```haskell
type RefInt = [
    get = /[Int]
    set = /[Int Sig]
]
```

is generalized to:

```haskell
type (Ref X) = [
    get = /[X]
    set = /[X Sig]
]
```

Ref is a parametric type—it describes a family of types.
Generic Reference Cell Object

The constructor for generic reference cells follows:

```python
def ref (#X init:X) : (Ref X) =
    ( new contents:^X
        run contents!init
        [get = \[res:/X] =
            contents?v = (contents!v | res!v)
            set = \[v:X c:Sig] =
                contents?_ = (contents!v | c![[]])
        ]
    )

(Ref T) ≡ [get = /[/T] set = /[T Sig]]
```
Generic Reference Cell Object

A reference cell for integers and one for strings can be created as follows:

```plaintext
run ( val ref1 = (ref Int 0)
    val ref2 = (ref String "one")
    (ref1.set 5);
    (prNL (int.toString (ref1.get)));
    (prNL (ref2.get));
    ()
)
```

If the type parameter is omitted, Pict will infer it if possible, e.g.:

- `(ref Int 0) ≡ (ref 0)`
- `(ref String "one") ≡ (ref "one")`