```
and true true
--- [1]: false
--- [2]: true
```

9.4 NATURAL NUMBERS

Our definition of natural numbers is inspired by the Peano natural numbers. We use two constructors

- zero is a natural number, written as Z;
- the successor of a natural number is a natural number, written as S;

and the BNF we defined when discussing how to encode inductive data.

```
0    = \s.\z.z
succ = \n.\s.\z.s (n s z)
```

This definition of `0` is trivial: given a successor function and a zero, return zero. The successor function seems more complex, but it uses the same underlying idea: given a number, a successor and a zero, apply the successor to the interpretation of that number using the same successor and zero.

We can then name some natural numbers as

```
1 = succ 0
2 = succ 1
3 = succ 2
4 = succ 3
5 = succ 4
6 = succ 5
...
```

even if we can not define an infinite number of terms as we might wish. The interpretation the natural number n as a higher order function is a function taking an argument `f` and applying them n times over the second argument.

```
5 not true
4 not true
double = \n.\s.\z.n (compose s s) z
double 3
--- [1]: false
--- [2]: true
--- [3]: 6
```

Addition $n + m$ applies the successor m times to n; and multiplication nm applies the n-fold application of the successor m times to 0.

```
plus = \m.\n.\s.\z.m s (n s z)
mult = \m.\n.\s.\z.m (n s) z
plus 2 1
mult 2 4
--- [1]: 3
--- [2]: 8
```

9.5 THE PREDECESSOR FUNCTION AND PREDICATES ON NUMBERS

The predecessor function is much more complex than the previous ones. As we can see, it is not trivial how could we compute the predecessor using the limited form of induction that Church numerals allow.

Stephen Kleene, one of the students of Alonzo Church only discovered how to write the predecessor function after thinking about it for a long time (and he only discovered it while a long visit at the dentist's, which is the reason why this definition is often called the ***wisdom tooth trick***, see [Cro75]). We will use a slightly different version of the definition that does not depend on a pair datatype.

We will start defining a *reverse composition* operator, called `rcomp`; and we will study what happens when it is composed to itself; that is

```
rcomp = \f.\g.\h.h (g f)
\f.3 (inc f)
\f.4 (inc f)
\f.5 (inc f)
--- [1]: λa.λb.λc.c (a (a (b a)))
--- [2]: λa.λb.λc.c (a (a (a (b a))))
--- [3]: λa.λb.λc.c (a (a (a (a (b a)))))
```

will allow us now to use the `b` argument to discard the first instance of the `a` argument and return the same number wihtout the last constructor. Thus, our definition of `pred` is

```
pred = \n.\s.\z.(n (inc s) (\x.z) (\x.x))
```

From the definition of `pred`, some predicates on numbers can be defined. The first predicate will be a function distinguishing a successor from a zero. It will be user later to build more complex ones. It is built by appliying a `const false` function `n` times to a true constant. Only if it is applied `0` times, it will return a true value.

```
iszero = \n.(n (const false) true)
iszero 0
iszero 2
--- [1]: true
--- [2]: false
```

From this predicate, we can derive predicates on equality and ordering.

```
leq = \m.\n.(iszero (minus m n))
eq  = \m.\n.(and (leq m n) (leq n m))
```

9.6 LISTS AND TREES

We would need two constructors to represent a list: a `nil` signaling the end of the list and a `cons`, joining an element to the head of the list. An example of list would be

$$\texttt{cons 1 (cons 2 (cons 3 nil))}.$$

Our definition takes those two constructors into account

```
nil  = \c.\n.n
cons = \h.\t.\c.\n.(c h (t c n))
```

and the interpretation of a list as a higher-order function is its `fold` function, a function taking a binary operation and an initial element and appliying the operation repeteadly to every element on the list.

$$\texttt{cons 1 (cons 2 (cons 3 nil))} \overset{fold\ plus\ 0}{\longrightarrow} \texttt{plus 1 (plus 2 (plus 3 0))} = 6$$

The `fold` operation and some operations on lists can be defined explicitly as

```
fold = \c.\n.\l.(l c n)
sum  = fold plus 0
prod = fold mult 1
all  = fold and true
any  = fold or false
length = foldr (\h.\t.succ t) 0

sum (cons 1 (cons 2 (cons 3 nil)))
all (cons true (cons true (cons true nil)))
--- [1]: 6
--- [2]: true
```

The two most commonly used particular cases of fold and frequent examples of the functional programming paradigm are `map` and `filter`.

- The **map** function applies a function `f` to every element on a list.
- The **filter** function removes the elements of the list that do not satisfy a given predicate. It *filters* the list, leaving only elements that satisfy the predicate.

They can be defined as follows.

```
map    = \f.(fold (\h.\t.cons (f h) t) nil)
filter = \p.(foldr (\h.\t.((p h) (cons h t) t)) nil)
```

On `map`, given a `cons h t`, we return a `cons (f h) t`; and given a `nil`, we return a `nil`. On `filter`, we use a boolean to decide at each step whether to return a list with a head or return the tail ignoring the head.

```
mylist = cons 1 (cons 2 (cons 3 nil))
sum (map succ mylist)
length (filter (leq 2) mylist)
--- [1]: 9
--- [2]: 2
```

Lists have been defined using two constructors and **binary trees** will be defined using the same technique. The only difference with lists is that the `cons` constructor is replaced by a `node` constructor, which takes two binary trees as arguments. That is, a binary tree is

- an empty tree; or
- a node, containing a label, a left subtree, and a right subtree.

Defining functions using a fold-like combinator is again very simple due to the chosen representation. We need a variant of the usual function acting on three arguments, the label, the right node and the left node.

```
-- Binary tree definition
node = \x.\l.\r.\f.\n.(f x (l f n) (r f n))
-- Example on natural numbers
mytree    = node 4 (node 2 nil nil) (node 3 nil nil)
triplesum = \a.\b.\c.plus (plus a b) c
mytree triplesum 0
--- [1]: 9
```

9.7 FIXED POINTS

A fixpoint combinator is a term representing a higher-order function that, given any function `f`, solves the equation

$$\texttt{x} = \texttt{f x}$$

for `x`, meaning that, if `fix f` is the fixpoint of `f`, the following sequence of equations holds

$$\texttt{fix}\, f = f(\texttt{fix}\, f) = f(f(\texttt{fix}\, f)) = f(f(f(\texttt{fix}\, f))) = \ldots$$

Such a combinator actually exists; it can be defined and used as

```
fix := (\f.(\x.f (x x)) (\x.f (x x)))
fix (const id)
--- [1]: id
```

Where `:=` defines a function without trying to evaluate it to a normal form; this is useful in cases like the previous one, where the function has no normal form. Examples of its applications are a *factorial* function or a *fibonacci* function, as in

```
fact := fix (\f.\n.iszero n 1 (mult n (f (pred n))))
fib  := fix (\f.\n.iszero n 1 (plus (f (pred n)) (f (pred (pred n)))))
fact 3
fib 3
--- [1]: 6
--- [2]: 5
```

Note the use of `iszero` to stop the recursion.

The fix function cannot be evaluated without arguments into a closed form, so we have to delay the evaluation of the expression when we bind it using `!=`. Our evaluation strategy, however, will always find a way to reduce the term if it is possible, as we saw in Corollary 1; even if it has intermediate irreducible terms.

```
fix              -- diverges
true  id fix     -- evaluates to id
false id fix     -- diverges
```

Other examples of the interpreter dealing with non terminating functions include infinite lists as in the following examples, where we take the first term of an infinite list without having to evaluate it completely or compare an infinite number arising as the fix point of the successor function with a finite number.

```
-- Head of an infinite list of zeroes
head = fold const false
```

```
head (fix (cons 0))
-- Compare infinity with other numbers
infinity != fix succ
leq infinity 6
---- [1]: 0
---- [2]: false
```

These definitions unfold as

- `fix (cons 0) = cons 0 (cons 0 (cons 0 ...))`, an infinite list of zeroes;
- `fix succ = succ (succ (succ ...))`, an infinite natural number.

10

PROGRAMMING IN THE SIMPLY TYPED λ-CALCULUS

This section explains how to use the simply typed λ-calculus to encode compound data structures and proofs in intuitionistic logic. We will use the interpreter as a typed language and, at the same time, as a proof assistant for the intuitionistic propositional logic.

This presentation of simply typed structures follows the Mikrokosmos tutorial and the previous sections on simply typed λ-calculus. All the code on this section is valid Mikrokosmos code.

10.1 FUNCTION TYPES AND TYPEABLE TERMS

Types can be activated with the commmand `:types on`. If types are activated, the interpreter will infer the principal type of every term before its evaluation. The type will then be displayed after the result of the computation.

Example 5 (Typed terms on Mikrokosmos). The following are examples of already defined terms on lambda calculus and their corresponding types. It is important to notice how our previously defined booleans have two different types; while our natural numbers will have all the same type except from zero, whose type is a generalization on the type of the natural numbers.

```
id    --- [1]: λa.a ⇒ id, I, ifelse :: A → A
true  --- [2]: λa.λb.a ⇒ K, true :: A → B → A
false --- [3]: λa.λb.b ⇒ nil, 0, false :: A → B → B
0     --- [4]: λa.λb.b ⇒ nil, 0, false :: A → B → B
1     --- [5]: λa.λb.(a b) ⇒ 1 :: (A → B) → A → B
2     --- [6]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
S     --- [7]: λa.λb.λc.((a c) (b c)) ⇒ S :: (A → B → C) → (A → B) → A → C
K     --- [8]: λa.λb.a ⇒ K, true :: A → B → A
```

If a term is found to be non-typeable, Mikrokosmos will output an error message signaling the fact. In this way, the evaluation of λ-terms that could potentially not terminate is prevented. Only typed λ-terms will be evaluated while the option `:types` is on; this ensures the termination of every computation on typed terms.

Example 6 (Non-typeable terms on Mikrokosmos). Fixed point operators are a common example of non typeable terms. Its evaluation on untyped λ-calculus would not terminate; and the type inference algorithm fails on them.

```
fix
--- Error: non typeable expression
fix (\f.\n.iszero n 1 (plus (f (pred n)) (f (pred (pred n))))) 3
--- Error: non typeable expression
```

Note that the evaluation of compound λ-expressions where the fixpoint operators appear applied to other terms can terminate, but the terms are still non typeable.

10.2 PRODUCT, UNION, UNIT AND VOID TYPES

Until now, we have only used the function type. That is to say that we are working on the implicational fragment of the simply-typed lambda calculus we described on the first . We are now going to extend our type system in the same sense we extended that simply-typed lambda calculus. The following types are added to the system

Type	Name	Description
→	Function type	Functions from a type to another
`MatchLowercase×`	Product type	Cartesian product of types
`+`	Union type	Disjoint union of types
⊤	Unit type	A type with exactly one element
⊥	Void type	A type with no elements

And the following typed constructors are added to the language,

Constructor	Type	Description
`(-,-)`	`A → B → A × B`	Pair of elements
`fst`	`(A × B) → A`	First projection
`snd`	`(A × B) → B`	Second projection
`inl`	`A → A + B`	First inclusion
`inr`	`B → A + B`	Second inclusion
`caseof`	`(A + B) → (A → C) → (B → C) → C`	Case analysis of an union
`unit`	⊤	Unital element
`abort`	`⊥ → A`	Empty function
`absurd`	`⊥ → ⊥`	Particular empty function

which correspond to the constructors we described on previous sections. The only new term is the `absurd` function, which is only a particular case of `abort`, useful when we want to make

explicit that we are deriving an instance of the empty type. This addition will only make the logical interpretation on the following sections clearer.

Example 7 (Extended STLC on Mikrokosmos). The following are examples of typed terms and functions on Mikrokosmos using the extended typed constructors. The following terms are presented

- a function swapping pairs, as an example of pair types;
- two-case analysis of a number, deciding whether to multiply it by two or to compute its predecessor;
- difference between `abort` and `absurd`;
- example term containing the unit type.

```
:load types
swap = \m.(snd m,fst m)
swap
--- [1]: λa.((SND a),(FST a)) ⇒ swap :: (A × B) → B × A
caseof (inl 1) pred (mult 2)
caseof (inr 1) pred (mult 2)
--- [2]: λa.λb.b ⇒ nil, 0, false :: A → B → B
--- [3]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
\x.((abort x),(absurd x))
--- [4]: λa.((ABORT a),(ABSURD a)) :: ⊥ → A × ⊥
```

Now it is possible to define a new encoding of the booleans with an uniform type. The type `⊤ + ⊤` has two inhabitants, `inl ⊤` and `inr ⊤`; and they can be used by case analysis.

```
btrue = inl unit
bfalse = inr unit
bnot = \a.caseof a (\a.bfalse) (\a.btrue)
bnot btrue
--- [1]: (INR UNIT) ⇒ bfalse :: A + ⊤
bnot bfalse
--- [2]: (INL UNIT) ⇒ btrue :: ⊤ + A
```

With these extended types, Mikrokosmos can be used as a proof checker on first-order intuitionistic logic by virtue of the Curry-Howard correspondence.

10.3 A PROOF IN INTUITIONISTIC LOGIC

Under the logical interpretation of Mikrokosmos, we can transcribe proofs in intuitionistic logic to λ-terms and check them on the interpreter. The translation between logical propositions and types is straightforward, except for the **negation** of a proposition $\neg A$, that must be written as $(A \longrightarrow \bot)$, a function to the empty type.