

Algorithmic with OCaml



Chapter 0.

Presentation

0.1. The programming language

OCaml (formerly Objective Caml) is a statically typed, garbage collected, general purpose programming language. OCaml especially emphasis software correctness via expressive static types. OCaml features both a friendly REPL (Read-Eval-Print-Loop) interpreter for programming in the small, and a batch compiler for programming in the large. OCaml uses type inference to alleviate the burden of type annotation while still offering both powerful modularity and excellent execution speed. From an historical perspective OCaml is a dialect of the ML family. ML is the language of choice for the implementation of theorem provers. That's a wonderful assurance. Because if ocaml is good enough for the Coq theorem prover there is great chances that it is also good enough for your own project no matter how ambitious or modest.

0.2. The book license

This book is a free culture work published under Creative Commons Attribution Share-Alike ([CC-BY-SA](https://creativecommons.org/licenses/by-sa/4.0/)) that everyone is welcome to contribute to. Another OCaml book under a [CC BY SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/) license is <https://ocamlbook.org/> (repository is <https://github.com/dmbaturin/ocaml-book>).

0.3. The motive

There already are plenty of OCaml blogs, tutorials and books so what makes this book somewhat special ? Well, i wanted to write a book about algorithms, i wanted to write a book about OCaml, i wanted to write a book about Coq and i wanted to write a book about Conceptual Graphs. So here it is, the most improbable programming book ever, with the cherry on the cake being a minimalist CGIF interpreter.

0.4. Installing ocaml from your package manager

This is not recommended because the distributed ocaml package is probably not up to date.

0.5. Installing ocaml from sources on Unix-likes

- Download the source archive <https://github.com/ocaml/ocaml/archive/4.13.0.zip>
- Extract the archive.
- Open a terminal at the extracted archive directory.
- Enter the following commands :

```
./configure
make world.opt
sudo -s      # login as a super user
umask 022    # make sure to give read & execute permission
make install
make clean
```

0.6. Installing ocaml from sources on Windows-Sub-Linux

- First install WSL from the Microsoft store and create a user account with a password.
- Your C: drive is /mnt/c in the bash console
- Then install make and gcc :

```
sudo apt update      # to update the repo packages
sudo apt install make # install gnu make
sudo apt install gcc  # install gcc
```
- Then continue as on a linux bash

0.7. Uninstalling ocaml when installed from sources

Enter the following commands :

```
rm -r /usr/local/lib/ocaml
rm /usr/local/bin/ocaml*
```

0.8. Installing ocaml from the opam package manager

This is the recommended way. See the opam site <https://opam.ocaml.org/>

0.9. What compiler command for the daily development ?

For software development you need to catch the most possible static and dynamic errors, so i would recommend something like this :

```
ocamlc.opt -w +A-ez-32 -c ...
```

If you just want to type-check one code file then the `-i` option alleviates the creation of object files :

```
ocamlc.opt -w +A-ez-32 -i ...
```

0.10. What compiler command for the project final release ?

For software release you need the best possible performance and compatibility, so i would recommend something like this :

```
ocamlopt.opt -noassert -unsafe -principal -o ...
```

You may add the `-nodynlink` switch if appropriate.

0.11. OCaml as a matter of interest

A friend (he coached me in Starcraft BroodWar protoss playing) and me on a French IRC chat :

He — I want to code a little video game. I know you are an ocaml programmer, what do you advocate ?

Me — ocaml + SDL2 bindings.

He — I dislike my ocaml university course. Show me a convincing ocaml code snippet.

Me — `type t = {n : 'a. 'a};;`

Me — `let make n = {n};;`

He — That's really weird ! Strong typing is just for performance anyway. What do you think about python ?

Me — The only very good python + SDL game is [SolarWolf](#).

His final choice was Lua + [LÖVE](#).

0.12. Thanks

This document owes much to multiple talented bloggers, forumers, book writers, code correctors and contributors that collectively make the amazing ocaml community. Many particular thanks go to Gabriel Scherer, Chris Okasaki, Matías Giovannini, Daniil Baturin, Craig Ferguson, Raphaël Proust, Ptival, dividee, Kiran Gopinathan, Armael and an anonymous contributor to [La Lettre de Caml](#) n°5 ... (please contact me if i have forgotten to credit your own contribution).

Chapter 1.

The functional type

1.1. The int type

OCaml is statically typed. Actually ocaml is almost type-obsessed hence you don't cheat ocaml at typing. It already shows when entering an integer in the REPL prompt :

```
# 1;;  
- : int = 1
```

Now you can't ignore that 1 is a int.

Of course the REPL prompt accepts many more arithmetic expression while respecting the precedence of operators :

```
# 1 + 2 * 3;;  
- : int = 7
```

Unfortunately not all arithmetic expression evaluate as an int, so ocaml warns you when something goes wrong :

```
# 1 / 0;;  
Exception: Division_by_zero.
```

The int range is not unlimited, it is exactly the interval [min_int,max_int] which is about half the size of a machine word.

1.2. Help me, my abs is negative !

```
# abs(min_int);;  
- : int = -4611686018427387904
```

This is not an OCaml bug. Any limited integer type in any programming language exhibits the same quirk. Don't suppose an abs must be positive !

1.3. The arrow type

Don't ignore the arrow type because it is the most important one. One could say being a functional language is to have the arrow type. The arrow type is an alternative name for the functional type, that is the type of functions. A function is a value of the arrow type. There are several ways to construct a new function. The `fun` keyword is one of them and it suffices to illustrate most functional idioms and properties. This code :

```
# fun n -> n + 1;;  
- : int -> int = <fun>
```

Creates a function that increments its argument.

Actually it's the exact same function as the predefined function `succ` :

```
# succ;;  
- : int -> int = <fun>
```

However we can't prove it because function equality is undecidable :

```
# (fun n -> n + 1) = succ;;  
Exception: Invalid_argument "equal: functional value".
```

The first thing to know about functions is that their variables are immutable. Actually, unless stated otherwise, all ocaml variables are immutable much like mathematical variables. Nevertheless you can assign a value to a variable, that is function application. The actual argument just follows the applied function.

```
# succ 1;;  
- : int = 2
```

It applies the integer 1 to the `succ` function by juxtaposition, no additional parenthesis is needed. An alternative way to apply an actual argument to a function is to swap positions (the actual argument is left, the applied function is right) and insert the `|>` operator between them :

```
# 1 |> succ;;  
- : int = 2
```

Otherwise the functional operator is in typical prefix position.

1.4. The arrow type rule

Let x be a value of the type T_X and f be a value of the arrow type $T_X \rightarrow T_F$ then x applied to f has type T_F .

Warning: x applied to f can eventually fail or loop forever. Thus a static type does not guarantee a dynamic actual value.

1.5. Infix operators

Another thing to know about functions is that any infix operator can be turned into a normal function by being parenthesized.

```
# (+);;  
- : int -> int -> int = <fun>
```

This is the exact same function as `fun a b -> a + b`.

```
# fun a b -> a + b;;  
- : int -> int -> int = <fun>
```

Warning: the `*` operator as a function must be written `(*)` otherwise ocaml would read it as an opening code comment. OCaml comments are nestable, they start with `(*` and end with `*)`. Likewise the `**` float operator as a function must be noted `(**)`.

1.6. Arrow associativity

The functional type `int -> int -> int` is the exact same type as `int -> (int -> int)`. Said otherwise, the arrow type constructor is right-associative. Or said otherwise `fun a b -> a + b` is the exact same function as `fun a -> (fun b -> a + b)`. Conversely `(+) a b` is the exact same value as `((+) a b)`. Said otherwise, ordinary function application is left-associative.

Finally, the `|>` operator is function application and thus is also left-associative.

1.7. Partial application

From a general function we can derive a more specialized function that accept one less argument.

Here is yet another synonym of `succ` :

```
# (+) 1;;  
- : int -> int = <fun>
```

This is not really surprising given that, because of right-associativity, the type of `(+)` is `int -> (int -> int)` and `(int -> int)` is the type of `succ`.

1.8. Over-application

Surprisingly enough it is also possible to apply more actual arguments than the declared formal arguments. This code applies 3 actual arguments to the identity function :

```
# (fun n -> n) (fun n -> n) (+) 1 2;;  
- : int = 3
```

1.9. The bool type

Many standard functions and operators (including = <> < <= >= >) have a bool result. The bool type has two predefined values true and false.

1.10. The conditional expression

The if...then...else... expression allows to select between two expressions according to a boolean condition. if...then...else... is a switchable expression, it's not a statement.

1.11. The conditional expression rule

Let e_{COND} be a bool expression and let e_{TRUE} and e_{FALSE} be two expressions of type T, then the expression if e_{COND} then e_{TRUE} else e_{FALSE} has the type T.

1.12. Outermost value declarations

The let keyword binds a (eventually functional) value to a name.

```
let pi = 3.1415926535898;;  
let add = fun a b -> a + b;;  
let minimum = fun a b -> if a < b then a else b;;  
let maximum = fun a b -> if a > b then a else b;;
```

The let binding is not permanent, a bounded name can eventually be rebound by a newer let. As an example this code replaces the predefined abs function :

```
let abs = fun z -> if z >= 0 then z else -z;;
```

For the coder convenience both the fun keyword and the -> arrow can be omitted :

```
let add a b = a + b;;  
let minimum a b = if a < b then a else b;;
```



```
let maximum a b = if a > b then a else b;;
let abs z = if z >= 0 then z else -z;;
```

1.13. Outermost recursive value declarations

The `rec` keyword allow a function name to be used (that is be applied) in his own body.

Here are some simple recursive functions :

```
let rec power a n =
  if n=0 then 1 else a * power a (n-1);;

let rec binomial n p =
  if (p=0) || (p=n) then 1 else
    binomial (n-1) p + binomial (n-1) (p-1);;

let rec fibonacci n =
  if n < 2 then 1 else fibonacci (n-1) + fibonacci (n-2);;
```

Imagine a frog that can climb either 1 or 2 steps in a single jump, then `fibonacci n` is how many ways this frog can climb by `n` steps. As presented here `fibonacci` is pretty slow because it enumerates all possible jump sequences. Later, at paragraph 1.17, we will present a much faster iterative version.

1.14. Mutually recursive functions

The `let rec ... and ...` expression allows multiple functions to be used in each-other bodies.

```
let rec odd n = if n=0 then false else even (n-1)
and even n = if n=0 then true else odd (n-1);;
```

1.15. Why not allow all functions to be recursive ?

Consider this code :

```
let rec id x = x
and f x = x && id true
and g x = x + id 1;;
```

It fails with an obscure type error.

Now consider the same code without mutual `rec` :

```
let id x = x
let f x = x && id true
```

```
let g x = x + id 1;;
```

Now it compiles without any type error. As a rule of thumb avoid unnecessary complexity everywhere possible otherwise you may get a more obscure type error than strictly needed.

1.16. Local value declarations

`let v = e in E` computes the expression `E` where all occurrences of the variable `v` are replaced by the expression `e`. Local variables are used either to split a large expression into smaller chunks or to factorize common sub-expressions. In the power function we don't want `power a (n/2)` to be computed more than one time :

```
let rec power a n =  
  if n = 0 then 1  
  else  
    let an2 = power a (n/2) in  
    if n mod 2 = 0 then an2 * an2  
    else an2 * a * an2;;
```

Imagine if `a` were a matrix, our power function would still be quite an efficient implementation. The more complex the data the more we gain by factoring common sub-expressions.

1.17. Mutually recursive local values

The `let rec ... and ... in ...` expression allows to declare local mutually recursive values, including functional values.

```
# let series n =  
  let rec u n = if n=0 then 1 else 2 * v (n-1)  
    and v n = if n=0 then 1 else 3 + u (n-1)  
  in u n;;  
val series : int -> int = <fun>  
# series 5;;  
- : int = 26
```

1.18. Accumulator variables

If function arguments are immutable like in mathematics then how could they be augmented ? They are augmented by applying new arguments using old values. How do we iterate ? We iterate by recursive application. An argument that is augmented by a recursive application is called an accumulator.

```

let rec factorial acc n =
  if n = 0 then acc
  else factorial (n * acc) (n - 1)

let factorial = factorial 1

let rec power acc a n =
  if n=0 then acc
  else power (acc * a) a (n - 1)

let power = power 1

let rec fibonacci a b n =
  if n < 2 then b
  else fibonacci b (a + b) (n - 1)

let fibonacci = fibonacci 1 1;;

```

The recursive step is not systematically $n - 1$, it's just something smaller than n , so $n/2$ is equally good and even better (because it decreases faster) :

```

let rec power a n =
  if n = 0 then 1
  else if n = 1 then a
  else
    let a2 = power (a * a) (n/2) in
    if n mod 2 = 0 then a2 else a * a2;;

```

1.19. Tail recursion

Tail-recursion is a stack optimization that allows a recursive function to use constant stack space (by recycling the current stack frame) instead of linear stack space (by allocating a new stack frame for each recursive call). The simplest form of tail-recursion-optimization is using accumulator variable(s) in order to place the recursive-application at the execution final position. Tail-recursion is the functional equivalent to procedural loops.

Our odd and even functions are tail-recursive because ocaml allows mutually tail-recursive functions.

The functions `factorial`, `power` and `fibonacci` in the previous paragraph are also tail-recursive. The `power` function is a special case, a kind of hybrid. First you have to inline `a2` because ocaml won't do it for you :

```

let rec power a n =
  if n = 0 then 1
  else if n = 1 then a

```

```

else
  if n mod 2 = 0 then power (a * a) (n/2)
  else a * power (a * a) (n/2);;

```

The then branch (when n is even) is tail-recursive (will be compiled as a goto) whereas the else branch (when n is even) is not (will be compiled as any application).

1.20. Partial functions

Most functions must be total functions. A total function is a function that always returns a result value. A partial functions is a function that may fail in a way or another. Wherever possible avoid partial functions such as `List.hd` and `List.tl`. Not every function can be total, so dealing with partial function is a necessary evil. However when doing evil do it good. There are basically two evil options.

The first option is to assert a condition.

`assert` has two advantages :

- It can enforce a structural invariant that the static type system can't capture. That helps debugging.
- It has zero user run-time cost because the `-noassert` compiler option removes them.

```

let rec factorial acc n =
  if n = 0 then acc
  else factorial (n * acc) (n - 1)

```

```

let factorial n =
  assert(n >= 0);
  factorial 1 n

```

The second option is to raise an exception.

1.21. Fixpoint combinators

Due to OCaml being rooted in λ -calculus and bicartesian-closed-categories you will eventually encounter programmer code that can be very hard to decipher without serious background. Here is one such wizardry. `zfix` is called a fixpoint combinator. It allows recursive applications in functions without `rec` :

```

let rec zfix f x = f (zfix f) x

```

```

let factorial fn n =
  if n = 0 then 1

```

```
    else n * fn (n - 1)
```

```
let factorial = zfix factorial (* zfix is partially applied here *)
```

```
let fibonacci fn a b n =  
    if n < 2 then b  
    else fn b (a + b) (n - 1)
```

```
let fibonacci = zfix fibonacci 1 1 (* zfix is over-applied here *)
```

Chapter 2.

Algebraic datatypes

2.1. The float type

A float literal is any number with a dot (eventually followed by an exponent part).

```
# 2.21e9;;  
- : float = 2210000000.
```

Except ****** all float infix operators (+, -, *, /) are terminated by a dot. You can use the ****** power operator to compute any number root :

```
# 27. ** (1. /. 3.);;  
- : float = 3.
```

The negation float prefix operator is **~-**.

```
# (~-.);;  
- : float -> float = <fun>
```

2.2. Limited float precision

```
# 42.456 -. 42.;;  
- : float = 0.4560000000000000307
```

You are kidding me, it should be just 0.456 isn't it ? OCaml has a bug. Well, ocaml certainly has bugs but this is not one of them. It's just standard IEEE-754 floats, double precision. The typical solution to save the user mental sanity is to castrate the precision by using the `Printf.printf` function :

```
# Printf.printf "%.4f\n" (42.456 -. 42.);;  
0.4560  
- : unit =()
```

2.3. The pair type

A pair type is constructed using the ***** infix type operator :

```
type complex = float * float;;
```

type is the keyword to declare a new user type.

A pair value is constructed using the comma infix value operator :

```
# (1.,2.);;  
- : float * float = (1., 2.)
```

The : colon operator is required to explicit a declared type such as complex :

```
# (1.,2. : complex);;  
- : complex = (1., 2.)
```

A pair type or a pair value can be heterogeneous :

```
# (1,true);;  
- : int * bool = (1, true)
```

2.4. The tuple type

A tuple type is a generalized pair type and is constructed using multiple * infix type operators :

```
type vertex3D = float * float * float;;
```

A tuple value is constructed using multiple comma infix value operators :

```
# (1.,2.,3. : vertex3D);;  
- : vertex3D = (1., 2., 3.)
```

Just like a pair a tuple can also be heterogeneous.

Pair and tuples types are sometimes called product-types.

2.5. Formal arguments filtering pairs or tuples

```
# let add_pair (a,b) = a + b;;  
val add_pair : int * int -> int  
  
# let add_triple (a,b,c) = a + b + c;;  
val add_triple : int * int * int -> int  
  
# let add_complex (xa,ya:complex) (xb,yb:complex) =  
  (xa +. xb, ya +. yb : complex);;  
val add_complex : complex -> complex -> complex = <fun>
```

2.6. Let-in filtering pairs or tuples

```
let fractional x =  
  let (f,i) = modf x in f;;
```

2.7. Parenthesis elimination for pairs or tuples

As a let in declaration or as a final result value parenthesis are usually optional.

```
let fractional x =  
  let f,i = modf x in f;;  
  
let succ_pair (a,b) = a + 1, b + 1;;  
let succ_triple (a,b,c) = a + 1, b + 1, c + 1;;
```

2.8. Tuple comparison

Tuples are compared (by `Stdlib.compare`) in lexical order. That means the left-most value has precedence. If two left-most values are equal then the next equal-rank values matter. That continues until the right-most values are reached.

```
# (1,9) < (2,5);;  
- : bool = true  
# (3,9,7) < (3,5,1);;  
- : bool = false
```

2.9. The enumerated type

A new enumerated type is declared using the `|` infix operator separating multiple constructors :

```
type day =  
  | Monday  
  | Tuesday  
  | Wednesday  
  | Thursday  
  | Friday  
  | Saturday  
  | Sunday;;
```

Warning: a constructor first letter must be uppercase.

Enumerated types are sometimes called sum-types.

2.10. Enumerated type and comparison


```
type compared =  
  | Less | Equal | More;;
```

According to the `Stdlib.compare` function, constructors are sorted from less to more. That means `Less < Equal < More` holds.

And `Monday < Tuesday < Wednesday < Thursday < Friday < Saturday < Sunday` also holds.

2.11. Overriding a type

You can override a previously declared type by declaring it again as a user type.

Warning: don't override a common standard type, it could catch you.

```
type int = Int;;  
1 + Int;;  
Error: This expression has type int/1  
       but an expression was expected of type int/2
```

2.12. The algebraic datatypes

Algebraic datatypes are the bread and butter of functional programming. An algebraic datatype models a disjoint union set. Practically it is an enumeration of (optional) tuples.

```
type poker_hand =  
  | HighCard of int * int * int * int * int  
  | OnePair of int * int * int * int  
  | TwoPair of int * int * int  
  | ThreeOfAKind of int  
  | Straight of int  
  | Flush of int * int * int * int * int  
  | FullHouse of int * int  
  | FourOfAKind of int  
  | StraightFlush of int
```

With the following card values :

```
# let ace,king,queen,jack = 50,40,30,20;;  
val ace : int = 50  
val king : int = 40  
val queen : int = 30  
val jack : int = 20
```

Tuples values are from the strongest card value to the weakest card value.

Now a poker hand can be compared to another poker hand :

```

HighCard(king, queen, jack, 10, 7) >
HighCard(king, queen, jack, 10, 5);;
-: bool = true
OnePair(queen, king, jack, 10) > OnePair(queen, king, jack, 7);;
-: bool = true
FullHouse(queen, jack) > FullHouse(queen, 10);;
-: bool = true
StraightFlush(8) > StraightFlush(6);;
-: bool = true

```

2.13. Recursive algebraic datatypes

Recursive types are allowed. You can use a type inside its own definition. Doing that we can declare the type of arithmetic expressions :

```

type arithmetic =
  | Int of int
  | Neg of arithmetic
  | Add of arithmetic * arithmetic
  | Sub of arithmetic * arithmetic
  | Mul of arithmetic * arithmetic
  | Div of arithmetic * arithmetic
;;

```

Recursive algebraic datatypes are also named inductive types. Inductive types play a central role in the semantic specification of linguistic-like components such as math expressions, math functions, logics and programs. Strangely enough (on the opposite of functions), recursive types do not require the `rec` keyword. Actually there exist a `nonrec` keyword to be used when you really want to declare a non-recursive type.

2.14. Pattern matching an algebraic datatype using match

The datatype `arithmetic` can be deconstructed using a `match ... with` followed by a disjoint set of equations.

```

# let rec eval expr =
  match expr with
  | Int n -> n
  | Neg a -> - eval a
  | Add(a,b) -> eval a + eval b
  | Sub(a,b) -> eval a - eval b
  | Mul(a,b) -> eval a * eval b
  | Div(a,b) -> eval a / eval b
;;
val eval : arithmetic -> int = <fun>

```

2.15. Understanding pattern matching

The canonical use of pattern matching is splitting an inductive value like `expr` into a disjoint set of equations.

Each equation has two sides, one left of the arrow and the resulting value right of the arrow.

The left side is a filter that introduces new variables to be used in the right side. This filter is made of two kinds of identifiers :

- an identifier starting with an uppercase letter is a constructor name.
- an identifier starting with a lowercase letter is a fresh new variable bounded to any value at his position.

The underscore character is a wildcard that acts much like an anonymous variable.

2.16. Pattern matching an algebraic datatype using function

Actually we don't even use the `expr` variable in `eval`.

So we can use `function` instead of `match`. `function` can filter an anonymous variable.

```
# let rec eval = function
| Int n -> n
| Neg a -> - eval a
| Add(a,b) -> eval a + eval b
| Sub(a,b) -> eval a - eval b
| Mul(a,b) -> eval a * eval b
| Div(a,b) -> eval a / eval b
;;
val eval : arithmetic -> int = <fun>
```

2.17. The pattern matching expression rule

Just like there is a rule for conditional expressions, there is a rule for pattern matching expressions.

This pattern matching rule is : all the equation right-members must have the same type.

2.18. The pitfalls of pattern matching

One cool feature of pattern matching is that it's extended to constant literals. Especially useful is character intervals that allow simple lexing functions.

```
# let alpha = function
| 'a'..'z' -> true
```

```

    | 'A'..'Z' -> true
    | c -> false;;
val alpha : char -> bool = <fun>

```

On the opposite a literal integer can be a bad idea.

```

# let rec factorial = function
  | n -> n * factorial (n - 1)
  | 0 -> 1;;
Warning 11: this match case is unused.

```

That warning 11 is because equations are ordered. First pattern is tested firstly, second pattern is tested secondly, and so on. Hence the new variable n bounds to any integer and the zero case is never reached. Pattern matching is overkill when you just need an integer conditional. The canonical factorial instead is conditional :

```

# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;

```

Another pitfall of pattern matching is the double use of the same new variable supposedly enforcing an equality constraint.

```

# let rec binomial = function
  | n,0 -> 1
  | n,n -> 1
  | n,p -> binomial (n-1,p) + binomial (n-1,p-1);;
Error: Variable n is bound several times in this matching

```

The n,n pattern does not mean two equal values, instead it means two different variables with the same identifier. Again the safe canonical binomial is not a pattern matching but a basic conditional :

```

# let rec binomial (n,p) =
  if p = 0 || n = p then 1
  else binomial (n-1,p) + binomial (n-1,p-1);;

```

Another common error in pattern matching is uncompleted equation set.

```

# let sign = function
  | Less -> -1
  | More -> +1;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Equal

```

You just forgot what if the `Equal` constructor applies.

Don't worry you will get a warning 8 each time a pattern matching is not exhaustive.

2.19. I have removed those semi-colons, how do i compile now ?

Typically you have entered some elaboration of the following code in the REPL

```
let print_bool = function
  | true  -> print_string "true"
  | false -> print_string "false";;

print_bool true;;
```

And everything is fine. Now it's time to compile, you remove the semi-colons and you get an error :

```
| false -> print_string "false"
```

Error: This function has type string -> unit
It is applied to too many arguments; maybe you forgot a `;'.

The explanation is that newlines is read as a blank space by the compiler so you are trying to compile the expression `print_string "false" print_bool true`. And the suggested remedy is to reintroduce some semi-colon(s). Don't listen the compiler message. What the compiler really expects is some declaration, something beginning with the `let`, `type`, `module`, `exception`, `class`... keyword. The following code snippet satisfies both the interpreter and the compiler :

```
let print_bool = function
  | true  -> print_string "true"
  | false -> print_string "false"

let () = print_bool true
```

2.20. A formal function deriver

Using a simple inductive type we can declare the type of $\mathbb{R} \rightarrow \mathbb{R}$ functions :

```
type function_x =
  | X          (* the x variable *)
  | R of float (* a real number  *)
  | Sin of function_x
  | Cos of function_x
  | Tan of function_x
  | Log of function_x
  | Exp of function_x
```

```

| Power of function_x * float
| Add of function_x * function_x
| Mul of function_x * function_x
;;

```

More $\mathbb{R} \rightarrow \mathbb{R}$ functions can be defined using only `function_x`.

`Sub(a,b)` is much like `Add(a,Mul(R(-1),b))`.

`Div(a,b)` is much like `Mul(a,Power(b,-1))`.

So we don't need `Sub` and `Div` because we want a disjoint union.

Let's introduce two utility functions that factorize constant multiplicand :

```

let factor = function
| R(1.),u -> u
| u,v -> Mul(u,v)

let product = function
| R(a),Mul(R(b),u) -> factor(R(a*.b),u)
| Mul(R(a),u),R(b) -> factor(R(a*.b),u)
| Mul(R(a),u),Mul(R(b),v) -> factor(R(a*.b),Mul(u,v))
| u,Mul(R(k),v) -> Mul(R(k),Mul(u,v))
| Mul(R(k),u),v -> Mul(R(k),Mul(u,v))
| u,v -> factor(u,v)
;;

```

Now the formal derivation of $\mathbb{R} \rightarrow \mathbb{R}$ functions :

```

let rec derive = function
| X -> R(1.)
| R(k) -> R(0.)
| Add(u,R(k)) -> derive(u)
| Add(u,v) -> Add(derive(u),derive(v))
| Mul(R(k),X) -> R(k)
| Mul(R(k),u) -> product(R(k),derive(u))
| Mul(u,v) -> Add(product(derive(u),v),product(u,derive(v)))
| Sin(u) -> product(derive(u),Cos(u))
| Cos(u) -> product(R(-1.),product(derive(u),Sin(u)))
| Tan(u) -> product(derive(u),Power(Cos(u),-2.))
| Log(u) -> product(derive(u),Power(u,-1.))
| Exp(u) -> product(derive(u),Exp(u))
| Power(u,2.) -> product(R(2.),product(derive(u),u))
| Power(u,a) -> product(R(a),product(derive(u),Power(u,a-.1.)))
;;

```

Let's play with it.

The derivation of $(2x)^{\frac{1}{2}}$ is $(2x)^{-\frac{1}{2}}$.

```

# derive(Power(Mul(R(2.),X),0.5));;
- : function_x = Power (Mul (R 2., X), -0.5)

```

And the derivation of $3\cos^2(x^2-1)$ is $-12x.\sin(x^2-1).\cos(x^2-1)$.

```
# derive(Mul(R(3.),Power(Cos(Add(Power(X,2.),R(-1.))),2.))));;
- : function_x =
Mul (R (-12.),
  Mul (Mul (X, Sin (Add (Power (X, 2.), R (-1.))),
    Cos (Add (Power (X, 2.), R (-1.)))))
```

2.21. Nested pattern matching

Most languages in the ML family have a companion theorem prover. OCaml is no exception. Coq is OCaml's companion theorem prover. One little syntax departure Coq has is the `match...with...end` pattern matching. Coq pattern matching is scoped and we will now see why this is a good decision.

```
type lexpr =
  (* lisp code *)
  | Var of int
  | Abs of lexpr
  | App of lexpr * lexpr
  | Let of lexpr * lexpr
  (* lisp values *)
  | Fun of (lexpr -> lexpr)
  | Int of int

let rec eval expr env =
  match expr with
  | Var n      -> List.nth env n
  | Abs body   -> Fun (fun x -> eval body (x::env))
  | App (f,arg) ->
      match eval f env with
      | Fun f -> f (eval arg env)
      | _ -> invalid_arg "Can't apply, not a function"
  | Let (e,body) -> eval body (eval e env::env)
  | _ -> expr
```

This code raises 3 warnings. This is because ocaml doesn't read the code indentation. OCaml can't tell our nested match must be properly scoped like that :

```
( match eval f env with
| Fun f -> f (eval arg env)
| _ -> invalid_arg "Can't apply, not a function" )
```

Once the nested match is correctly parenthesized our code is the sketch of a minimalist lisp-like interpreter.

2.22. Dealing with partial functions using the result type

Our formal derivation is a total function.

But our evaluator for arithmetic is a partial function :

```
# eval(Div(Int 1, Int 0));;  
Exception: Division_by_zero.
```

Can we make it a total function ? The answer is yes we can.

First we introduce 3 utility functions :

```
let ok1 v f =  
  match v with  
  | Ok x -> Ok (f x)  
  | Error _ -> v  
  
let ok2 v1 v2 f =  
  match v1,v2 with  
  | Ok x1,Ok x2 -> Ok (f x1 x2)  
  | Error _, _ -> v1  
  | _, Error _ -> v2  
  
let result2 v1 v2 f =  
  match v1,v2 with  
  | Ok x1,Ok x2 -> f x1 x2  
  | Error _, _ -> v1  
  | _, Error _ -> v2  
;;
```

Now we use these utilities to propagate the error until we obtain a total function :

```
# let rec eval = function  
  | Int n ->  
    Ok n  
  | Neg a ->  
    ok1 (eval a) (~-)  
  | Add(a,b) ->  
    ok2 (eval a) (eval b) (+)  
  | Sub(a,b) ->  
    ok2 (eval a) (eval b) (-)  
  | Mul(a,b) ->  
    ok2 (eval a) (eval b) ( * )  
  | Div(a,b) ->  
    result2  
      (eval a) (eval b)  
      (fun a b -> if b=0 then Error Division_by_zero  
                  else Ok (a/b))  
;;
```



```
val eval : arithmetic -> (int, exn) result = <fun>
```

2.23. Mutually recursive datatypes

Just like `let rec ... and ...` allows mutual recursive values, `type ... and ...` allows mutually recursive types.

```
type tree =  
  | Node of int * forest  
and forest =  
  | Leaf of int  
  | Forest of tree * forest
```

Chapter 3.

Module as namespaces

3.1. Component granularity

We know types. We know functions that manipulates types. Functions and types must be tied together, that's what is called component programming. In Object-Oriented-Programming the basic component is the class. Because classes are too fine-grain, in OCaml the basic component is the module.

3.2. Module components

Now how do we tie function_x type and derive function ?

Of course we want both a white-box version and a black-box version.

The black-box is what the component user sees, the white-box is actual code.

The black-box version will only export a type `t` and the `derive : t -> t` function. The white-box version will have all this plus our factor and product utility functions. OCaml has several possibilities to achieve this but we will only present one. One possibility is to put all code in one single file `FormalDerivationX.ml`:

```
module type Type =
sig
  type t =
    | X          (* the x variable *)
    | R of float (* a real number *)
    | Sin of t
    | Cos of t
    | Tan of t
    | Log of t
    | Exp of t
    | Power of t * float
    | Add of t * t
    | Mul of t * t
  val derive : t -> t
end

module Data : Type =
struct
  type t = ...
  let factor = ...
```

```

    let product = ...
    let rec derive = ...
end

```

The source file can be read in the REPL by the `#mod_use` directive :

```
#mod_use "FormalDerivationX.ml";;
```

But now `t` becomes `FormalDerivationX.Data.t` and `derive` becomes `FormalDerivationX.Data.derive`. Don't be afraid of very long names. Long names makes things unique. Long names are not the plague, long names are the cure. Moreover long module names are easy to shorten. The rest of this chapter is about the many ways ocaml offers you to shorten long names.

3.3. Opened modules

There are two options for opening a module, the gentle and the extreme.

The gentle is `open FormalDerivationX`

Now `t` becomes `Data.t` and `derive` becomes `Data.derive`. Quite reasonable.

The extreme is `open FormalDerivationX.Data`

Now `t` is `t` and `derive` is `derive`. Can't be simpler.

Probably even too simple, you may want the global namespace to be as clean as possible.

One solution is a locally opened module.

```
# let open FormalDerivationX.Data in
  derive(Mul(R(3.),Power(Cos(Add(Power(X,2.),R(-1.))),2.))));;
- : FormalDerivationX.Data.t =
...

```

Or even shorter :

```
# FormalDerivationX.Data.
(derive(Mul(R(3.),Power(Cos(Add(Power(X,2.),R(-1.))),2.))));;
- : FormalDerivationX.Data.t =
...

```

3.4. Module aliasing

Another popular usage is module aliasing. Module aliasing is tailor-made to bring long names and short identifiers together.

```
module FX = FormalDerivationX.Data;;
```

See how the formal derivation of $3\cos^2(x^2-1)$ is still quite terse.

```
# FX.derive(FX.Mul(FX.R(3.),FX.Power(FX.Cos(FX.Add(FX.Power(FX.X,2.),FX.R(-1.))),2.))));
- : FX.t =
FX.Mul (FX.R (-12.),
  FX.Mul (FX.Mul (FX.X, FX.Sin (FX.Add (FX.Power (FX.X, 2.), FX.R (-1.)))),
    FX.Cos (FX.Add (FX.Power (FX.X, 2.), FX.R (-1.)))))
```

3.5. Local modules

Another way to keep the global namespace clean is a local module.

```
# let module FX = FormalDerivationX.Data in
FX.derive(FX.Mul(FX.R(3.),FX.Power(FX.Cos(FX.Add(FX.Power(FX.X,2.),FX.R(-1.))),2.))));
- : FormalDerivationX.Data.t =
...
```

Chapter 4.

Binary trees and the art of sorting

4.1. The passion trees

Trees attract functional programmers like magnets attract iron. A tree is much like a list with multiple (at least two) tails. Often a functional tree is used as a persistent replacement for an Abstract Data Type such as a priority queue, a dynamic array, a stack, a binary-heap and so on.

4.2. The binary tree inductive type

One of the most fruitful inductive type is the binary tree. The binary tree is either Empty or a Fork with a recursive left branch, an item, and a recursive right branch. We declare it with `int` as the item type, so that our algorithmic approach is not cluttered up with domain values and operations.

```
type t =  
  | Empty  
  | Fork of t * int * t
```

Depending on it's final role, a binary tree type exports the `empty`, `singleton`, `add`, `member`, `remove`, `union` values and operations, or a subset or a superset. One fundamental thing to remember is that two items can be compared. In some ways we are reinventing the art of sorting but the functional way. Please don't skip this chapter if you have good knowledge of imperative sorting because it's a whole different world. At the moment the only thing we can do is to code the `empty` and the `singleton n` values.

```
let empty =  
  Empty  
  
let is_empty t =  
  t = Empty  
  
let singleton n =  
  Fork(Empty, n, Empty)
```

The binary tree code often uses conventional variable names.

- `l` is the left branch

- r is the right branch
- m, n are elements / items
- la is the left branch of the ta tree
- na is the element / item of the ta tree
- ra is the right branch of the ta tree
- lb is the left branch of the tb tree
- nb is the element / item of the tb tree
- rb is the right branch of the tb tree
- i is an index natural number
- acc is an accumulator
- f is a function

A good binary tree is balanced. Unbalanced binary trees perform bad. While there is no definitive function to test a binary tree balance, there is a measure known as the Strahler number. The Strahler number gives the depth of the biggest complete binary tree embedded in a particular binary tree.

```
let rec strahler = function
| Empty -> 0
| Fork(l,n,r) ->
    let sl = strahler l and sr = strahler r in
    if sl = sr then sl+1 else max sl sr
```

4.3. The binary search tree set, the member & add operations

The invariant of a binary search tree is that everything in the left branch is strictly lesser than the item, and everything in the right branch is strictly greater than the item. That suffices to code the `member` n operation : if the tree is empty then return `false`, otherwise if n is lesser than the item then recursively explore the left branch, if n is greater than the item then recursively explore the right branch, else n is equal to the item then return `true`.

```
let rec member n = function
| Empty -> false
| Fork(l,m,r) ->
    if n < m then member n l
    else if n > m then member n r
    else true
```

That also suffices to code the `add` n operation : if the tree is empty then return singleton n , otherwise if n is lesser than the item then recursively add n in the left branch, if n is greater than the item then recursively add n in the right branch, else n is equal to the item then return the initial tree.

```

let rec add n t =
  match t with
  | Empty -> singleton n
  | Fork(l,m,r) ->
    if n < m then Fork(add n l,m,r)
    else if n > m then Fork(l,m,add n r)
    else t

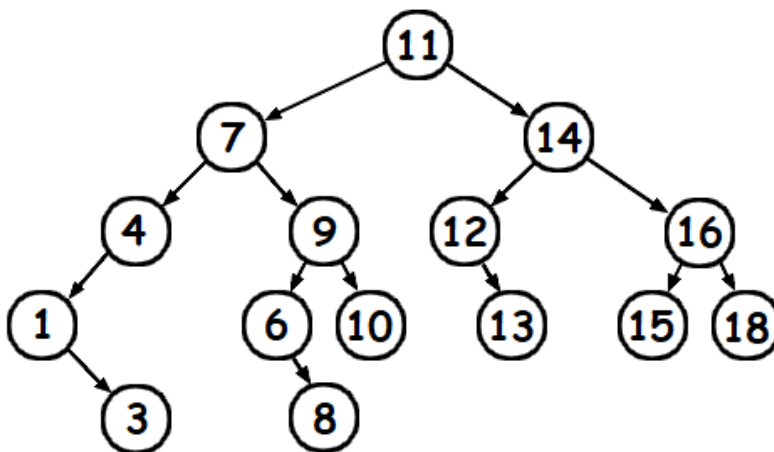
```

Remark that the t tree as the last argument allows us to pipe the set operations :

```

let my_set =
  singleton 11
  |> add 7 |> add 4 |> add 14 |> add 9 |> add 12 |> add 6 |> add 1
  |> add 8 |> add 13 |> add 3 |> add 16 |> add 10 |> add 15 |> add 18
  ;;

```



4.4. The binary search tree set, the remove operation

The remove operation first mimics the add operation : if the tree is empty then return Empty, otherwise if n is lesser than the item then recursively remove n in the left branch, if n is greater than the item then recursively remove n in the right branch. Else n is equal to the item then concat the left and right branches together.

```

let rec remove n = function
  | Empty -> Empty
  | Fork(l,m,r) ->
    if n < m then Fork(remove n l,m,r)
    else if n > m then Fork(l,m,remove n r)
    else concat l r

```

What is this unknown concat ta tb operation ? What does it do ? concat is a new constructor. First concat needs all elements in ta to be strictly lesser than

all elements in tb. Second concat does very little if one tree is empty. Third if both ta and tb aren't empty they are merged :

- ta is preserved as the left branch
- the minimum element of tb is promoted as the root element
- tb is deprived of it's minimum element and becomes the new right branch

```
let concat ta tb =
  match ta,tb with
  | _,Empty -> ta
  | Empty,_ -> tb
  | _,Fork(lb,nb,rb) ->
    Fork(ta,minimum nb lb,remove_minimum lb nb rb)
```

Finding the minimum element is just iterating along the left spine :

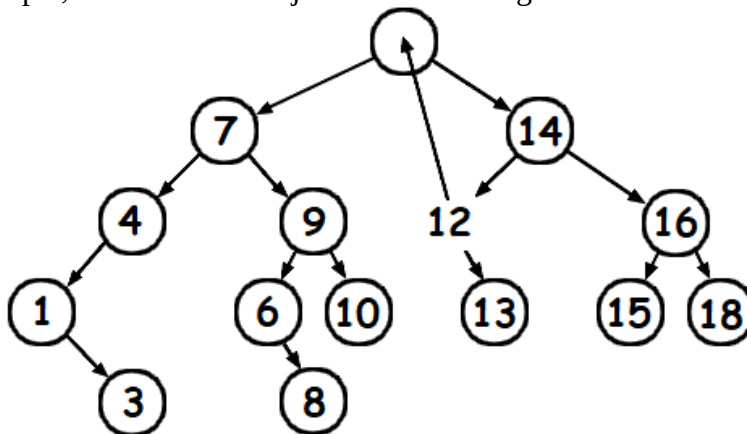
```
let rec minimum acc = function
  | Empty -> acc
  | Fork(l,n,r) -> minimum n l
```

Removing the minimum element is recursive style following the left spine :

```
let rec remove_minimum la na ra =
  match la with
  | Empty -> ra
  | Fork(lb,nb,rb) -> Fork(remove_minimum lb nb rb,na,ra)
```

Remark that the actual source code must declare these operations in the reverse order, first introduce minimum and remove_minimum, then concat then finally remove.

In our example, to remove 11 is just the same thing as to concat 7 14.



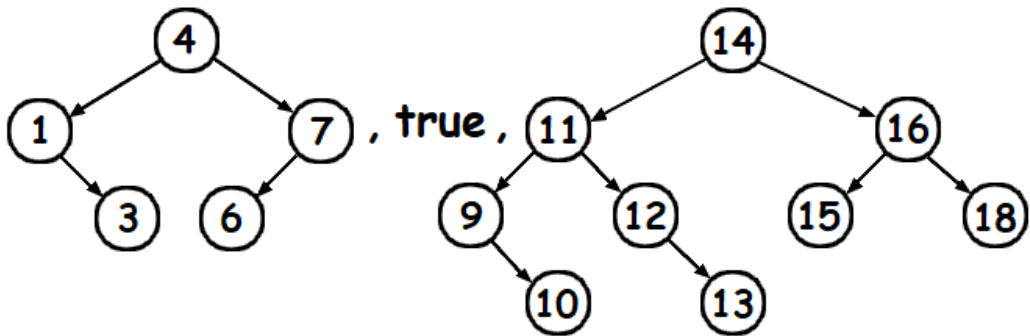
4.5. The binary search tree set, the union operation

Now we want the set union operation.

Before coding union we need the `split n` deconstructor that separates a set into a less than `n` set and a more than `n` set. As an accessory `split` also serves as a membership operation, we will need that later. The `split n` deconstructor structurally recurses on left or right branch depending on how the element compares to `n`.

```
let rec split n = function
| Empty -> Empty, false, Empty
| Fork(l,m,r) ->
    if n < m then
        let la,present,ra = split n l in la,present,Fork(ra,m,r)
    else if n > m then
        let lb,present,rb = split n r in Fork(l,m,lb),present,rb
    else l,true,r
```

`split 8 my_set;;`



The union operation deconstructs `tb` using `split na tb` and then `Fork` reconstructs by structural recursion on both left and right branches.

```
let rec union ta tb =
    match ta,tb with
    | _,Empty -> ta
    | Empty,_ -> tb
    | Fork(la,na,ra),_ ->
        let lb,_,rb = split na tb
        in Fork(union la lb,na,union ra rb)
```

4.6. The binary search tree set, more set operations

Now we want the set cardinal and the remaining set operations.

The cardinal operation is easy to write recursively :

```
let rec cardinal = function
  | Empty -> 0
  | Fork(l,_,r) -> cardinal l + 1 + cardinal r
```

However there is a less obvious alternative that is more iterative. Let's accumulate all along the entire binary tree :

```
let rec cardinal acc = function
  | Empty -> acc
  | Fork(l,_,r) -> cardinal (cardinal acc r + 1) l
let cardinal =
  cardinal 0
```

That seems contrived at first but it can greatly enhance the performance of serializing operations.

The intersection and difference operations first deconstruct using `split na tb` and then, depending on the membership result, `Fork` or `concat` reconstruct by structural recursion on both left and right branches.

```
let rec intersection ta tb =
  match ta,tb with
  | _,Empty | Empty,_ -> Empty
  | Fork(la,na,ra),_ ->
    let lb,present,rb = split na tb in
    if present then Fork(intersection la lb,na,intersection ra rb)
    else concat (intersection la lb) (intersection ra rb)

let rec difference ta tb =
  match ta,tb with
  | _,Empty -> ta
  | Empty,_ -> Empty
  | Fork(la,na,ra),_ ->
    let lb,present,rb = split na tb in
    if present then concat (difference la lb) (difference ra rb)
    else Fork(difference la lb,na,difference ra rb)
```

The disjoint predicate first deconstructs using `split na tb` and then, depending on the membership result, returns `false` or structurally recurses on both left and right branches.

```
let rec disjoint ta tb =
  match ta,tb with
  | _,Empty | Empty,_ -> true
  | Fork(la,na,ra),_ ->
    let lb,present,rb = split na tb in
```

```

    if present then false
    else disjoint la lb && disjoint ra rb

```

The subset predicate structurally recurses if $na = nb$ otherwise it recurses on carefully chosen parts.

```

let rec subset ta tb =
  match ta,tb with
  | Empty,_ -> true
  | _,Empty -> false
  | Fork(la,na,ra),Fork(lb,nb,rb) ->
    if na < nb then
      subset (Fork(la,na,Empty)) lb && subset ra tb
    else if na > nb then
      subset (Fork(Empty,na,ra)) rb && subset la tb
    else
      subset la lb && subset ra rb

```

Set equality is reciprocal inclusion :

```

let equal ta tb =
  subset ta tb && subset tb ta

```

4.7. The binary search tree set, the filter operation

Now we want to filter a set according to a cond predicate.

We deconstruct using function and, depending on cond n, we Fork or concat reconstruct by structural recursion on both left and right branches.

```

let rec filter cond = function
  | Empty -> Empty
  | Fork(l,n,r) ->
    if cond n then Fork(filter cond l,n,filter cond r)
    else concat (filter cond l) (filter cond r)

```

4.8. The binary search tree set, the interval operation

Now we want to extract the subset delimited by the [low,high] interval. We recurse on the left branch or the right branch or both depending on how the element compares to the limits.

```

let rec interval low high = function
  | Empty -> Empty
  | Fork(l,n,r) ->
    if high < n then interval low high l
    else if low > n then interval low high r
    else Fork(interval low high l,n,interval low high r)

```

4.9. The binary search tree set, the to_list operation

Here the connection between binary tree & sorting becomes more obvious as we convert the binary tree to a sorted list. This is straightforward recursively :

```
let rec to_list = function
  | Empty -> []
  | Fork(l,n,r) -> to_list l @ [n] @ to_list r
```

However this structurally recursive version is too slow and memory hungry so we rather adopt the iterative style already used for the cardinal operation.

```
let rec to_list acc = function
  | Empty -> acc
  | Fork(l,n,r) -> to_list (n::to_list acc r) l
let to_list =
  to_list []
```

4.10. The binary search tree set, conclusion

If the binary search tree is balanced enough then the performance is quite good, that is about $O(\log \text{cardinal})$ amortized case for the most basic operations. An imperative hash-table does dictionary operations in about amortized $O(1)$. We have seen however that a binary search tree can do much more than only dictionary operations. Moreover many variants (like Splay tree, AVL tree, Red-Black tree) do faster dictionary operations than vanilla binary search tree without compromising set operations.

4.11. The Braun tree

The Braun tree is a binary tree popularized by Chris Okasaki. The invariant of a Braun tree is that every left branch has at most one more item than the right branch. Because a Braun tree looks much like a complete binary tree, the path from the root to a leaf is guaranteed to have minimal length which is an excellent performance testimony.

Now we want the Braun tree size. And we want it faster than enumerating all items.

The code is from the paper [Three Algorithms on Braun Trees by Chris Okasaki](#).

```
let rec diff n = function
  | Empty -> if n = 0 then 0 else assert false
  | Fork(l,_,r) ->
    if n = 0 then 1
    else if n mod 2 = 1 then diff ((n - 1) / 2) l
    else diff ((n - 2) / 2) r
```

```

let rec size = function
| Empty -> 0
| Fork(l,_,r) -> let m = size r in 2 * m + 1 + diff m l

```

This function computes in $O(\log^2 \text{size})$.

4.12. The Braun stack, the add operation

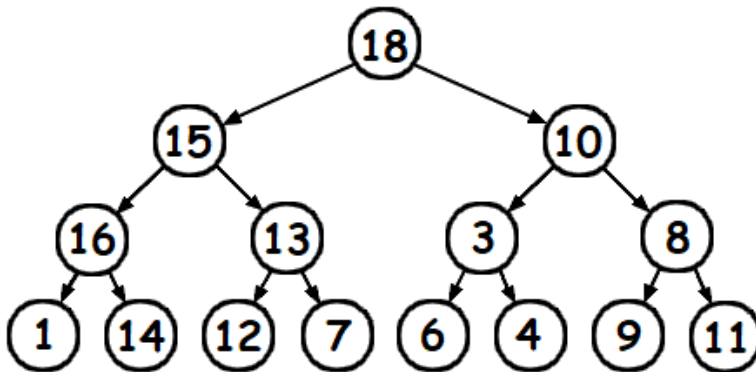
A Braun tree can implement a Braun stack. The insertion of a new item in a Braun stack uses a unique scheme : the left branch and the right branch are swapped with the new item being recursively inserted in the left branch.

```

let rec add n = function
| Empty -> singleton n
| Fork(l,m,r) -> Fork(add m r,n,l)

let my_Braun_stack =
  singleton 11
  |> add 7 |> add 4 |> add 14 |> add 9 |> add 12 |> add 6 |> add 1
  |> add 8 |> add 13 |> add 3 |> add 16 |> add 10 |> add 15 |> add 18
;;

```



4.13. The Braun stack, the member operation

Now we want to know the last added item or any older item.

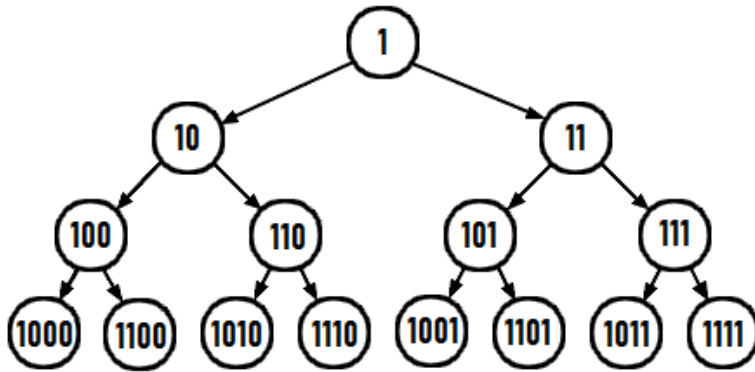
The last item will be member 0, the last but 1 will be member 1, and so on ...

```

let rec member i = function
| Empty -> invalid_arg "BraunStack.member"
| Fork(l,n,r) ->
  if i = 0 then n else
  if i land 1 = 1 then member (i / 2) l
  else member (i / 2 - 1) r

```

The path from root to an item is encoded in its index number. First $i + 1$ is converted to base 2, then, from the less significant bit to the most significant bit, each 0 digit means go left and each 1 digit means go right.



4.14. The Braun stack, the remove operation

Now we want to remove the last added item.

```

let remove = function
| Empty -> invalid_arg "BraunStack.remove"
| Fork(l,_,r) -> concat l r

```

What is this unknown `concat ta tb` operation ? What does it do ? `concat` is a new constructor. First `concat` needs the Braun tree `ta` to have at most one more item than the Braun tree `tb`. Second `concat` does very little if `ta` is empty. Third if `ta` is not empty then `ta` and `tb` are merged :

- `tb` becomes new the left branch
- `concat l r` becomes the new right branch

```

let rec concat ta tb =
  match ta with
  | Empty -> Empty
  | Fork(l,n,r) -> Fork(tb,n,concat l r)

```

4.15. The Braun stack, the replace operation

The `replace i x` operation binds the member `i` item to the value `x`.

```

let rec replace i x = function
| Empty -> invalid_arg "BraunStack.replace"
| Fork(l,y,r) ->
  if i = 0 then Fork(l,x,r) else
  if i land 1 = 1 then Fork(replace (i / 2) x l,y,r)
  else Fork(l,y,replace (i / 2 - 1) x r)

```

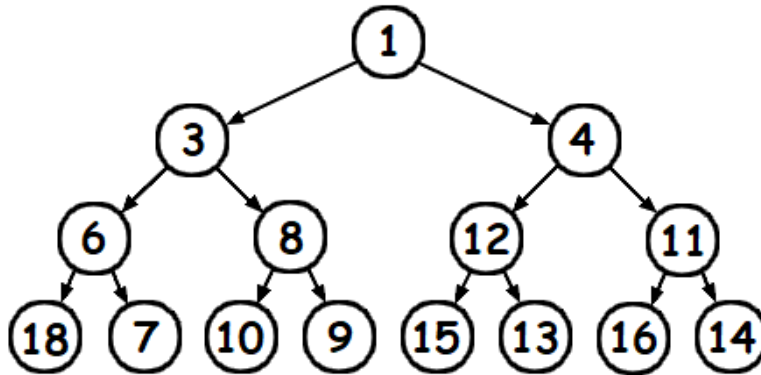
4.16. The Braun min heap, the add operation

A Braun tree can also implement a Braun heap. We choose a min-heap rather than a max-heap. The invariant of a min-heap is that the root item is lesser than any item in its branches.

The left branch and the right branch are swapped with the new item being recursively inserted in the left branch or becoming the new root item depending on the comparison.

```
let rec add n = function
| Empty -> singleton n
| Fork(l,m,r) ->
    if n < m then Fork(add m r,n,l)
    else Fork(add n r,m,l)

let my_Braun_min_heap =
    singleton 11
|> add 7 |> add 4 |> add 14 |> add 9 |> add 12 |> add 6 |> add 1
|> add 8 |> add 13 |> add 3 |> add 16 |> add 10 |> add 15 |> add 18
;;
```



4.17. The Braun min heap, the member operation

Now we want to retrieve the least item of the heap. It just sits at the tree root.

```
let member = function
| Empty -> invalid_arg "BraunHeap.Data.member"
| Fork(_,n,_) -> n
```

4.18. The Braun min heap, the replace operation

Now we want to remove the minimum/root item and add a new item n in a single operation.

```
let rec replace n = function
| Empty -> invalid_arg "BraunHeap.replace"
| Fork((Fork(_,m,_) as l),_,Empty)
    when m < n ->
```

```

    Fork(replace n l,m,Empty)
| Fork((Fork(_,na,_) as l),_,(Fork(_,nb,_) as r))
    when na < n || nb < n ->
    if na < nb
    then Fork(replace n l,na,r)
    else Fork(l,nb,replace n r)
| Fork(l,_,r) -> Fork(l,n,r)

```

4.19. The Braun min heap, the remove operation

Now we want to only remove the minimum/root item.

```

let rec remove = function
| Empty -> invalid_arg "BraunHeap.remove"
| Fork(t,_,Empty) -> t
| Fork(Empty,_,_) -> assert false
| Fork((Fork (_,na,_) as l),_,(Fork( _,nb,_) as r)) ->
    if na < nb
    then Fork(r,na,remove l)
    else Fork(replace na r,nb,remove l)

```

4.20. The heap sort operation

We are not done yet with sorting.

Once you have a min-or-max heap your can sort its items using `to_list`.

```

(* heap to sorted list *)
let rec to_list t =
  if is_empty t then []
  else member t :: to_list (remove t)

```

4.21. The retrace max heap, the add operation

Now we want a max-heap that preserves its history. The invariant of a max-heap is that the root item is greater than any item in its branches.

```

let rec add n t =
  match t with
  | Empty -> singleton n
  | Fork(_,m,_) when n > m -> Fork(t,n,Empty)
  | Fork(l,m,Empty) -> Fork(l,m,singleton n)
  | Fork(l,m,(Fork(_,k,_) as r)) when k > n ->
    Fork(l,m,add n r)
  | Fork(l,m,r) -> Fork(l,m,Fork(r,n,Empty))

let my_retrace_max_heap =
  singleton 11
  |> add 7 |> add 4 |> add 14 |> add 9 |> add 12 |> add 6 |> add 1

```



```
|> add 8 |> add 13 |> add 3 |> add 16 |> add 10 |> add 15 |> add 18
;;
```

4.22. The retrace max heap, the member operation

Now we want to retrieve the greatest item of the heap. It just sits at the tree root.

```
let member = function
| Empty -> invalid_arg "RetraceHeap.member"
| Fork(_,n,_) -> n
```

4.23. The retrace max heap, the remove operation

Now we want to remove the maximum/root item.

If the left branch is empty then we return the right branch and if the right branch is empty then we return the left branch. Otherwise we recursively Fork reconstruct depending on the comparison of n_a and n_b .

```
let rec remove = function
| Empty -> invalid_arg "RetraceHeap.remove"
| Fork(Empty,n,Empty) -> Empty
| Fork(l,n,Empty) -> l
| Fork(Empty,n,r) -> r
| Fork((Fork(la,na,ra) as l),n,(Fork(lb,nb,rb) as r)) ->
  if nb > na then
    let ll = remove (Fork(l,n,lb))
    in Fork(ll,nb,rb)
  else
    let rr = remove (Fork(ra,n,r))
    in Fork(la,na,rr)
```

4.24. The retrace max heap, the retrace operation

Now we want to recover the history of add operations. We already know this code, it's just binary tree linearization.

```
let rec retrace acc = function
| Empty -> acc
| Fork(l,n,r) -> to_list (n::retrace acc r) l
let retrace =
  retrace []
```

4.25. The retrace max heap, the to_list operation

Now suppose we have a max-heap and yet still want items to be sorted in increasing order.

An iterative version of the heap sort allows us to do that.

```
(* iterative heap to sorted list *)
let rec to_list acc t =
  if is_empty t then acc
  else to_list (member t :: acc) (remove t)
let to_list =
  to_list []
```

The same applies if we want a min-heap sorted in decreasing order.

4.26. The binary tree, conclusion

We have seen the type `t = Empty | Fork of t * int * t` being enriched enough to implement many different abstract data types. More importantly we have seen the binary tree to be the advocated way to sort data. Sorting data is even more important than you think. Appropriately sorted data is often the prime avenue to optimal performance.

4.27. The binary tree as a graph data type

Instead of implementing more heaps (Skew-heap and Leftist-heap would be good candidates) we totally change direction and implement directed graph (whom vertices are labeled with `int`) in an unexpected way.

Let's begin with the streamlined module interface.

```
module DirectedGraph
:
sig
  type t
  val empty : t
  val add : int -> int -> t -> t
  val member : int -> int -> t -> bool
  val successors : int -> t -> int list
  val predecessors : int -> t -> int list
  val subgraph : t -> t -> bool
  val equal : t -> t -> bool
end
=
struct (* ... *)
```

4.28. The directed graph, the new type `t`

It is just another binary tree where the edge's source and destination are stored at the nodes.

```

type t =
  | Empty
  | Fork of t * int * int * t
let empty =
  Empty

```

4.29. The directed graph, the add operation

The edges (x,y) are added one by one starting from the empty graph.

The x and y alternate roles. At first x is the source vertex and y is the destination vertex. However they will be swapped at each step down.

add x y g constructs a new graph depending on the comparison of x and u :

- if the g graph is empty then return a singleton
- else if x is less than u then the edge (y,x) is added to the left branch
- else if x is greater than u then the edge (y,x) is added to the right branch
- otherwise x = u
- if also y = v then the edge is already in the graph g
- otherwise the edge (y,x) is added to the right branch

(* add an edge (x,y) to the graph *)

```

let rec add x y = function
  | Empty -> Fork(Empty,x,y,Empty)
  | Fork (l,u,v,r) as g ->
    if x < u then Fork (add y x l,u,v,r)
    else if x > u then Fork (l,u,v,add y x r)
    else if y = v then g
    else Fork (l,u,v,add y x r)

```

```

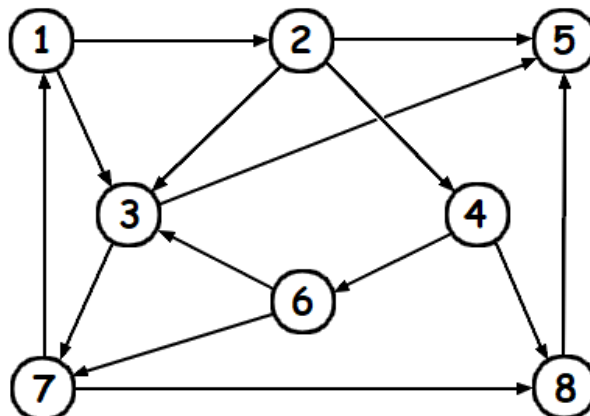
let my_directed_graph =

```

```

  empty
  |> add 6 3 |> add 2 4 |> add 1 2 |> add 8 5 |> add 1 3
  |> add 4 6 |> add 2 3 |> add 7 8 |> add 2 5 |> add 7 1
  |> add 4 8 |> add 3 5 |> add 6 7 |> add 3 7
  ;;

```



4.30. The directed graph, the member operation

As usual, the membership predicate follows the same scheme as insertion, but a bool is returned instead.

- if the graph is empty then return false
- if x is less than u then the edge (y,x) is to be found in the left branch
- if x is greater than u then the edge (y,x) is to be found in the right branch
- otherwise $x = u$
- if also $y = v$ then return true
- otherwise the edge (y,x) is to be found in the right branch

```
(* is the edge (x,y) in the graph ? *)
let rec member x y = function
| Empty -> false
| Fork (l,u,v,r) ->
    if x < u then member y x l
    else if x > u then member y x r
    else if y = v then true
    else member y x r
```

4.31. The directed graph, the successors & predecessors operations

We have first to deal with the non-discriminating levels. Once again there is the recursive style and the iterative style.

```
(* recursive version *)
(* deal with the non-discriminating levels *)
let apply f w = function
| Empty -> []
| Fork (l,u,v,r) ->
    f w r @ (if w = v then [u] else []) @ f w r

(* iterative version *)
(* deal with the non-discriminating levels *)
let apply f w acc = function
| Empty -> acc
| Fork (l,u,v,r) ->
    f w (f w (if w = v then u::acc else acc) r) l
```

We use the apply iterative version. Now we deal with the discriminating levels.

```
(* deal with the discriminating levels *)
let rec loop x acc = function
| Empty -> acc
| Fork (l,u,v,r) ->
    if x < u then apply loop x acc l
    else if x > u then apply loop x acc r
```

```
else v::apply loop x acc r
```

Now the main difference between successors and predecessors is whether we start with the discriminating levels or the non-discriminating ones.

```
(* successors & predecessors *)
let successors x =
  loop x []
let predecessors y =
  apply loop y []
```

4.32. The directed graph, the subgraph & equality operations

With the help of the member predicate the local mutually recursive functions sub1 and sub2 close the deal.

```
(* is ga a subgraph of gb ? *)
let subgraph ga gb =
  let rec sub1 = function
    | Empty -> true
    | Fork (l,u,v,r) -> member u v gb && sub2 l && sub2 r
  and sub2 = function
    | Empty -> true
    | Fork (l,u,v,r) -> member v u gb && sub1 l && sub1 r
  in sub1 ga
```

Graph equality is reciprocal inclusion :

```
(* graph equality *)
let equal ga gb =
  subgraph ga gb && subgraph gb ga
```

4.33. The ternary search tree set

A ternary tree is much like a list with three tails. It uses lexicographical order, that means it is used to store lists and sequences. Much like a binary tree it has a less branch and a more branch, the new branch is the equal branch.

Let's begin with the module interface :

```
module TernarySearchTree
:
sig
  type t
  val empty : t
  val add : string -> t -> t
  val member : string -> t -> bool
```

```

    val remove : string -> t -> t
    val prefix : string -> t -> t
    val to_list : t -> string list
end
=
struct (*...*)

```

The variable name conventions are :

- lt is the less than branch
- eq is the equal branch
- gt is the greater than branch
- s is a string
- i is an index number

4.34. The ternary search tree set, the new type t

A ternary search tree is much similar to a binary search tree except now we have 3 branches. We will use it to store char sequences, that is strings.

```

type t =
| End
| Path of char * t * t * t
| Item of char * t * t * t

```

```

let empty =
    End

```

- the End constructor means a string end
- the Path constructor means a discriminating char
- the Item constructor means a string member of the set

4.35. The ternary search tree, the member operation

We begin with the member operation because it is somewhat simpler than the add operation.

```

let rec member s i = function
| End -> false
| Path (c, lt, eq, gt) ->
    if s.[i] < c then member s i lt
    else if s.[i] > c then member s i gt
    else member s (i+1) eq
| Item (c, lt, eq, gt) ->
    if s.[i] < c then member s i lt
    else if s.[i] > c then member s i gt
    else if i + 1 = String.length s then true

```

```
    else member s (i+1) eq
```

```
let member s =  
  member s 0
```

4.36. The ternary search tree, the add operation

The add operation borrows much code and structure from the member operation plus it inserts the needed constructors.

```
let rec add s i = function  
  | End ->  
    if i + 1 = String.length s then  
      Item (s.[i],End,End,End)  
    else  
      Path (s.[i],End,add s (i+1) End,End)  
  | Path (c,lt,eq,gt) ->  
    if s.[i] < c then Path (c,add s i lt,eq,gt)  
    else if s.[i] > c then Path (c,lt,eq,add s i gt)  
    else if i + 1 = String.length s then Item (c,lt,eq,gt)  
    else Path (c,lt,add s (i+1) eq,gt)  
  | Item (c,lt,eq,gt) as t ->  
    if s.[i] < c then Item (c,add s i lt,eq,gt)  
    else if s.[i] > c then Item (c,lt,eq,add s i gt)  
    else if i + 1 = String.length s then t  
    else Item (c,lt,add s (i+1) eq,gt)  
  
let add s =  
  add s 0  
  
let my_ternary_search_tree =  
  empty  
  |> add "January" |> add "February" |> add "March"  
  |> add "April" |> add "May" |> add "June" |> add "July"  
  |> add "August" |> add "September" |> add "October"  
  |> add "November" |> add "December"  
  ;;
```

4.37. The ternary search tree, the remove operation

The remove operation borrows much code and structure from the add operation.

```
let rec remove s i = function  
  | End ->  
    invalid_arg "TernarySearchTree.remove"  
  | Path (c,lt,eq,gt) ->  
    if s.[i] < c then Path (c,remove s i lt,eq,gt)  
    else if s.[i] > c then Path (c,lt,eq,remove s i gt)  
    else Path (c,lt,remove s (i+1) eq,gt)
```

```

| Item (c,lt,eq,gt) ->
  if s.[i] < c then Item (c,remove s i lt,eq,gt)
  else if s.[i] > c then Item (c,lt,eq,remove s i gt)
  else if i + 1 = String.length s then Path (c,lt,eq,gt)
  else Item (c,lt,remove s (i+1) eq,gt)

let remove s =
  remove s 0

```

4.38. The ternary search tree, the prefix operation

The prefix s function returns the internal sub-tree that contains all string beginning with s.

```

let rec prefix s i = function
| End -> End
| Path (c,lt,eq,gt) | Item (c,lt,eq,gt) ->
  if s.[i] < c then prefix s i lt
  else if s.[i] > c then prefix s i gt
  else if i + 1 = String.length s then eq
  else prefix s (i+1) eq
let prefix s =
  prefix s 0

```

4.39. The ternary search tree, the to_list operation

The ternary search tree is clearly yet another sorted abstract data type. Hence we can write a to_list function that returns a sorted list of char sequences. Of course there are two styles, the recursive style and the iterative style.

```

let append s c =
  s ^ String.make 1 c

(* the recursive version *)
let rec to_list s = function
| End -> []
| Path (c,lt,eq,gt) ->
  to_list s lt @ to_list (append s c) eq @ to_list s gt
| Item (c,lt,eq,gt) -> let asc = append s c in
  to_list s lt @ [asc] @ to_list asc eq @ to_list s gt

let to_list = to_list ""

(* the iterative version *)
let rec to_list acc s = function
| End -> acc
| Path (c,lt,eq,gt) ->
  to_list (to_list (to_list acc s gt) (append s c) eq) s lt
| Item (c,lt,eq,gt) -> let asc = append s c in

```



```
    to_list (to_list (asc::to_list acc s gt) asc eq) s lt  
let to_list = to_list [] ""
```

Chapter 5.

Certifying your code with Coq

5.1. This chapter is optional

Quality matters. However if you feel certification is overkill or too much demanding then you can jump directly to the chapter 6.

5.2. Why certify ?

Because the only alternative is rigorous unit tests. Whom of tests or certification is a lesser pain is up to you to decide.

5.3. How much certification can be trusted ?

Of course if the specifications are partial or inexact then the code can be as well. Otherwise the provided proofs are a guaranty that the code meets the specification. Testing becomes a thing of the past, the Jurassic world of trusted components.

5.4. Is Coq for doing maths or for doing programming ?

You can do both maths and programming.

Of course it helps a lot if you have a math background (even modest) because Coq is all about the type `Prop`. The type `Prop` is the type of propositions, that is math properties, whereas the type `Set` is the type of program data. From a programmer perspective Coq reuse many ocaml concepts and syntax. From a math perspective coq has innovative features that drastically limit the number of explicit parameters. Because maths are more implicit than programming, coq has to reconcile being implicit and being fully formal. Moreover, because formal proofs are much about try and fails, coq has to reconcile being fully interactive and being error-inflexible. And Coq does a pretty good job at that.

5.5. coqtop

coqtop is the interactive command line interface.

```
damien@user:~# coqtop
Welcome to Coq 8.11.0 (March 2020)
```

```
Coq <
```

coqtop is interactive and is good enough for small projects. The main coqtop pitfall is that you can't redefine something which is quite annoying. Also Coq has a pretty steep learning curve. Hence the first commands recommended to be learned are :

- **coqtop** because you have to start the session
- **Reset Initial.** because you have to restart the session from scratch
- **Restart.** because you have to restart the proof from scratch
- **Abort All.** because the proof may reveal to be more tricky than anticipated
- **Admitted.** because the next proof may be easier than the current one
- **Show Script.** because you are lost in your own proof
- **Quit.** because you are a genius or because you have to surrender

The proofs themselves are done using tactics on a kind of sequent calculus. Sequent calculus is not something a programmer wants to study by a book or documentation, it's more learning through practice.

5.6. Coq modules as a namespace

Coq has modules just like ocaml. They are more limited but more interactive. Coq interactivity is based on a keep-it-simple law : every sentence or command must end with a dot.

5.7. The Operation module

We are not building a full math framework with a Relation, Operation, Application module trilogy however a minimal Operation module is a nice commodity that will serve our first coq mini-project.

```
Module Operation.
```

This opens the Operation module declaration.

```
Section operations.
```

This opens a new section, that is a scope where some variables can be implicit.

```
Variable U : Set.
```

This declares a new variable `U` that will be implicit in the whole section.
Definition `Operation := U -> U -> U`.

This defines a new type `Operation`. Note that here `U` is an implicit type parameter.

```
Variable op  : Operation.    (* the law      *)
Variable e   : U.            (* the neutral *)
Variable inv : U -> U.       (* the inverse *)
```

This declares 3 new variables that can be implicit.

```
Definition associative : Prop :=
  forall x y z: U, op x (op y z) = op (op x y) z.
```

This declares what is the associativity property.

```
Definition left_neutral : Prop :=
  forall x: U, op e x = x.
```

```
Definition right_neutral : Prop :=
  forall x: U, op x e = x.
```

This declares what is a left neutral & right neutral element.

```
Definition neutral : Prop :=
  left_neutral /\ right_neutral.
```

This declares what is a neutral element.

```
Definition left_symmetric : Prop :=
  forall x: U, op (inv x) x = e.
```

```
Definition right_symmetric : Prop :=
  forall x: U, op x (inv x) = e.
```

This declares what is a left inverse & right inverse.

```
Definition symmetric_inverse : Prop :=
  left_symmetric /\ right_symmetric.
```

This declares what is an inverse.

```
End operations.
```

This closes the operations section.

```
End Operation.
```

This closes the Operation module.

The Operation module is now defined and we can print its internal type :

```
Print Module Operation.
```

5.8. The Monoid module-type

The monoid is a familiar programmer notion. The 'a list type equipped with the append law and the [] neutral is a monoid. \mathbb{N} equipped with the \times law and the 1 neutral is another monoid. This is why when you do the product of an empty int list you have to return 1 (because that preserves the monoid structure). There even exist an ocaml BatFingerTree module that returns an Abstract Data Type given a monoid as a parameter.

```
Module Type Monoid.
```

This opens the Monoid module-type declaration.

```
Import Operation.
```

This allows Operation code usage.

```
Parameter U : Set.  
Parameter law : Operation U.  
Parameter zero : U.
```

This defines 3 new parameters any monoid must have : a set, a law and a zero.

```
Axiom law_associative : associative U law.  
Axiom zero_neutral : neutral U law zero.
```

This introduces 2 new monoid axiom any monoid must satisfy : the law is associative and zero is a neutral element.

```
End Monoid.
```

This closes the Monoid module.

5.10. The $(\mathbb{N},+,0)$ monoid module

This is our first coq proof : we prove that \mathbb{N} equipped with the + law and the 0 neutral is a monoid.

```
Require Import Arith.Plus.
```

We import a coq library that contains basic facts about the nat type.

```
Module NatPlusZero : Monoid.
```

```
Import Operation.
```

We open the NatPlusZero module, we force it to be a Monoid and we use the Operation module.

```
Definition U := nat.  
Definition law := plus.  
Definition zero := 0.
```

This 3 definitions satisfy the 3 parameters obligations. Note that within coq the uppercase letter 0 means $zero \in \mathbb{N}$ and the uppercase letter S means succ.

```
Definition law_associative : associative U law.  
Proof.
```

Now we engage to prove that our law is associative.

```
unfold associative. intros.  
unfold law. rewrite plus_assoc.  
reflexivity.  
Qed.
```

Quod Erat Demonstrandum.

```
Definition zero_neutral : neutral U law zero.  
Proof.  
unfold neutral. split.  
unfold left_neutral.  
intros. unfold law. unfold zero.  
rewrite plus_0_l. reflexivity.  
unfold right_neutral.  
intros. unfold law. unfold zero.  
rewrite plus_0_r. reflexivity.  
Qed.
```

```
End NatPlusZero.
```

5.11. The Group module-type

A group is a monoid equipped with an inverse operation.

```
Module Type Group.
```

```
Include Monoid.
```

We reuse the Monoid module-type and will extend it.

```
Import Operation.
```

```
Parameter inverse : U -> U.
```

This defines 1 new parameter any group must have : an inverse operator.

```
Axiom inverse_symmetric : symmetric_inverse U law zero inverse.
```

This introduces 1 new axiom any group must satisfy : the inverse of an element is both a `left_inverse` and a `right_inverse`.

```
End Group.
```

```
Print Module Type Group.
```

That confirms that we actually have extended the monoid concept.

5.13. The `DirectProductGroup` module-function

The direct product of two groups H and K is noted $H \times K$.

We can build this product operator as a module-function, that is a module that takes modules H and K as arguments and returns a new module $H \times K$.

```
Module DirectProductGroup (H K : Group) : Group.
```

```
Import Operation.
```

```
Record product : Set := make {h : H.U; k : K.U}.
```

```
Definition U := product.
```

```
Definition law x y := make (H.law (h x) (h y)) (K.law (k x) (k y)).
```

```
Definition zero := make H.zero K.zero.
```

```
Definition inverse x := make (H.inverse (h x)) (K.inverse (k x)).
```

The definition of the direct product is done. All we need now is to prove `law_associative`, `zero_neutral` and `inverse_symmetric`.

```
Definition law_associative : associative U law.
```

```
Proof.
```

```
unfold associative. intros.
```

```

    unfold law. simpl.
    rewrite H.law_associative.
    rewrite K.law_associative.
    reflexivity.
    Qed.

Definition zero_neutral : neutral U law zero.
Proof.
  unfold neutral. split.
  unfold left_neutral.
    intros. unfold law. simpl.
    rewrite (proj1 H.zero_neutral).
    rewrite (proj1 K.zero_neutral).
    dependent inversion x. simpl. reflexivity.
  unfold right_neutral.
    intros. unfold law. simpl.
    rewrite (proj2 H.zero_neutral).
    rewrite (proj2 K.zero_neutral).
    dependent inversion x. simpl. reflexivity.
  Qed.

Definition inverse_symmetric : symmetric_inverse U law zero inverse.
Proof.
  unfold symmetric_inverse. split.
  unfold left_inverse.
    intros.
    unfold law. unfold inverse. simpl.
    rewrite (proj1 H.inverse_symmetric).
    rewrite (proj1 K.inverse_symmetric).
    reflexivity.
  unfold right_inverse.
    intros.
    unfold law. unfold inverse. simpl.
    rewrite (proj2 H.inverse_symmetric).
    rewrite (proj2 K.inverse_symmetric).
    reflexivity.
  Qed.

End DirectProductGroup.

```

We are done with coq and maths, now it's the turn of coq for programmers.

5.14. Coq as a program code certifier

Unfortunately Coq can't directly certify ocaml code. Actually coq has his own programming language called Gallina, and the gallina code can be certified with coq.

5.15. The Gallina language

The Gallina language is much like a small pure subset of ocaml.

The main differences are :

- the recursion must be well-funded
- the definitions use the `:=` operator instead of the `=` operator
- the standard libraries/functions differ from the ocaml ones
- the constructors are functional (arrows) instead of a tuple argument
- the match is scoped
- the match uses the `=>` arrow instead of the `->` arrow
- Gallina allows dependant types

5.16. The Gallina binary tree

First we define our inductive type.

```
(* a tree is either empty or a nat with left & right branches *)
Inductive tree : Set :=
| Empty: tree
| Fork: tree -> nat -> tree -> tree.
```

Now we can define the Strahler function :

```
(* here are defined nat basics *)
Require Import Arith.Arith_base.

(* strahler number *)
Fixpoint strahler t :=
  match t with
  | Empty => 0
  | Fork l n r =>
    let sl := strahler l in
    let sr := strahler r in
    if beq_nat sl sr then S sl else max sl sr
  end.
```

Unfortunately proving that `strahler t` is the depth of the biggest embedded complete binary tree in `t` is beyond the author's talent. Anyway we are more interested in the sorting capabilities of this tree type.

5.17. The Gallina binary search tree

We still have to paraphrase our ocaml code.

```
Inductive member : nat -> tree -> Prop :=
| root_member :
  forall l n r,
  member n (Fork l n r)
| left_member :
```

```

    forall m n l r,
    member n l ->
    member n (Fork l m r)
| right_member :
    forall m n l r,
    member n r ->
    member n (Fork l m r).

(* insert n in the tree t *)
Fixpoint add n t :=
  match t with
  | Empty => Fork Empty n Empty (* create a singleton *)
  | Fork l m r =>
    match n ?= m with
    | Eq => t (* n is already present *)
    | Lt => Fork (add n l) m r (* insert n in the left branch *)
    | Gt => Fork l m (add n r) (* insert n in the right branch *)
    end
  end.
end.

```

5.18. The Gallina binary search tree properties

Now we really enter the heart of the matter.

Of course it is desirable that once added an element becomes a member :

```

Theorem add_main_property :
  forall n t, member n (add n t).

```

Of course it is also desirable that adding an element doesn't erase anything else :

```

Theorem add_is_conservative :
  forall m n t, member m t -> member m (add n t).

```

Of course it is finally desirable that adding an element doesn't break the tree order :

```

Theorem add_preserves_order :
  forall t, tree_ordered t ->
  forall n, tree_ordered (add n t).

```

For some tree_ordered predicate that is still to be defined.

5.19. The Gallina binary search tree, add_main_property

```

(* nat_compare_eq is defined here *)
Require Import Arith.Compare_dec.

```

```

Theorem add_main_property :

```

```

forall n t, member n (add n t).
Proof.
  intros n t. induction t.
  (* Empty *)
  simpl.
  apply root_member.
  (* Fork *)
  simpl. remember (n ?= n0) as cmp.
  destruct cmp.
  (* Eq *)
  symmetry in Heqcmp.
  apply (nat_compare_eq n n0) in Heqcmp.
  subst n0. apply root_member.
  (* Lt *)
  apply left_member. exact IHt1.
  (* Gt *)
  apply right_member. exact IHt2.
Qed.

```

5.20. The Gallina binary search tree, add_is_conservative

```

Theorem add_is_conservative :
  forall m n t, member m t -> member m (add n t).
Proof.
  intros m n t H. induction t.
  (* Empty *)
  unfold add.
  apply left_member. exact H.
  (* Fork *)
  remember (Fork t1 n0 t2) as node.
  destruct H.
  (* root_member *)
  unfold add. destruct (n ?= n1).
  apply root_member.
  apply root_member.
  apply root_member.
  (* left_member *)
  inversion Heqnode. rewrite H1 in H.
  unfold add. destruct (n ?= n0).
  apply left_member. exact H.
  apply IHt1 in H. apply left_member. exact H.
  apply left_member. exact H.
  (* right_member *)
  inversion Heqnode. rewrite H3 in H.
  unfold add. destruct (n ?= n0).
  apply right_member. exact H.
  apply right_member. exact H.
  apply IHt2 in H. apply right_member. exact H.
Qed.

```

5.21. The Gallina binary search tree, the tree_ordered predicate

The `tree_ordered Empty` case is true.

The `tree_ordered (Fork l m r)` case is built using both `tree_less l m` and `tree_more r m`.

```
Inductive tree_less : tree -> nat -> Prop :=
| empty_tree_less :
  forall b, tree_less Empty b
| fork_tree_less :
  forall l m r b,
    tree_less l b -> tree_less r b -> m < b ->
    tree_less (Fork l m r) b.
```

```
Inductive tree_more : tree -> nat -> Prop :=
| empty_tree_more :
  forall a, tree_more Empty a
| fork_tree_more :
  forall l m r a,
    tree_more l a -> tree_more r a -> m > a ->
    tree_more (Fork l m r) a.
```

```
Inductive tree_ordered : tree -> Prop :=
| empty_tree_ordered :
  tree_ordered Empty
| fork_tree_ordered :
  forall l m r,
    tree_ordered l -> tree_ordered r ->
    tree_less l m -> tree_more r m ->
    tree_ordered (Fork l m r).
```

5.22. The Gallina binary search tree, member bounds

Fact `tree_less_upper_bound`:

`forall t n, tree_less t n <-> forall m, member m t -> m < n.`

Proof.

```
split.
(* -> *)
intros Hless m Hm. induction t.
inversion Hm.
inversion Hm; inversion Hless; subst; auto.
(* <- *)
intros H. induction t.
constructor.
constructor.
  apply IHt1. intros m Hm. apply H. apply left_member. assumption.
  apply IHt2. intros m Hm. apply H. apply right_member. assumption.
  apply H. apply root_member.
```

Qed.

Fact `tree_more_lower_bound`:

`forall t n, tree_more t n <-> forall m, member m t -> m > n.`

```

Proof.
  split.
  (* -> *)
  intros Hmore m Hm. induction t.
  inversion Hm.
  inversion Hm; inversion Hmore; subst; auto.
  (* <- *)
  intros H. induction t.
  constructor.
  constructor.
    apply IHt1. intros m Hm. apply H. apply left_member. assumption.
    apply IHt2. intros m Hm. apply H. apply right_member. assumption.
    apply H. apply root_member.
Qed.

```

5.23. The Gallina binary search tree, splitting members

```

Lemma member_left:
  forall l m r n, tree_ordered (Fork l m r) ->
  member n (Fork l m r) -> n < m -> member n l.
Proof.
  intros l m r n Hord Hmem Hlt.
  inversion Hord; subst; clear Hord.
  inversion Hmem; subst; clear Hmem.
  contradict Hlt. apply lt_irrefl.
  assumption.
  apply tree_more_lower_bound with (m:=n) in H5.
  contradict H5. apply gt_asym. assumption.
  assumption.
Qed.

```

```

Lemma member_right:
  forall l m r n, tree_ordered (Fork l m r) ->
  member n (Fork l m r) -> n > m -> member n r.
Proof.
  intros l m r n Hord Hmem Hlt.
  inversion Hord; subst; clear Hord.
  inversion Hmem; subst; clear Hmem.
  contradict Hlt. apply lt_irrefl.
  apply tree_less_upper_bound with (m:=n) in H4.
  contradict H4. apply lt_asym. assumption. assumption.
  assumption.
Qed.

```

5.24. The Gallina binary search tree, add_preserves_order

```

Lemma add_preserves_less:
  forall t n m, tree_less t n -> m < n -> tree_less (add m t) n.
Proof.
  induction t; intros; inversion H; subst; clear H.

```

```

    repeat constructor. assumption.
    simpl. destruct (m ?= n); constructor; auto.
Qed.

```

Lemma add_preserves_more:

```

  forall t n m, tree_more t n -> m > n -> tree_more (add m t) n.

```

Proof.

```

  induction t; intros; inversion H; subst; clear H.
  repeat constructor. assumption.
  simpl. destruct (m ?= n); constructor; auto.

```

Qed.

Theorem add_preserves_order :

```

  forall t, tree_ordered t ->
  forall n, tree_ordered (add n t).

```

Proof.

```

  induction t; intros.
  (* Empty *)
  simpl. repeat constructor.
  (* Fork *)
  simpl. remember (n0 ?= n) as cmp. symmetry in Heqcmp.
  destruct cmp.
  assumption.
  inversion H; subst; clear H. constructor; auto.
  apply nat_compare_lt in Heqcmp.
  apply add_preserves_less; assumption.
  inversion H; subst; clear H. constructor; auto.
  apply nat_compare_gt in Heqcmp.
  apply add_preserves_more; assumption.

```

Qed.

Chapter 7.

Polymorphic recursion

7.1. Polymorphic recursion without data

Polymorphic recursion can appear even in a simple recursive function :

```
# let rec f x y = f x x;;  
val f : 'a -> 'a -> 'b = <fun>
```

`f` is expected to have `'a->'b->'c` type but surprisingly has type `'a->'a->'b`. The problem is that the type of the `y` argument is not generalized enough. We can further generalize the type of the `y` argument by explicitly specifying `'b. 'a->'b->'c` as the type of `f`.

```
# let rec f : 'b. 'a -> 'b -> 'c = fun x y -> f x x;;  
val f : 'a -> 'b -> 'c = <fun>
```

7.2. Nested data types

Nested data types are the inductive types that require polymorphic recursion.

7.3. The DoubleDimension module

The `DoubleDimension` module is a type-polymorphic version of the `DirectedGraph` module.

It has the same operations but this time it stores `'a * 'b` values instead of the monomorphic `int * int` values.

```
module DoubleDimension  
:  
sig  
  type ('a,'b) t  
  val empty : ('a,'b) t  
  val add : 'a -> 'b -> ('a,'b) t -> ('a,'b) t  
  val member : 'a -> 'b -> ('a,'b) t -> bool  
  val first : 'b -> ('a,'b) t -> 'a list  
  val second : 'a -> ('a,'b) t -> 'b list  
  val for_all : ('a -> 'b -> bool) -> ('a,'b) t -> bool  
  val subset : ('a,'b) t -> ('a,'b) t -> bool
```

```

    val equal : ('a,'b) t -> ('a,'b) t -> bool
    val to_list : ('a,'b) t -> ('a * 'b) list
end
=
struct (*...*)

```

7.4. DoubleDimension, the new type ('a,'b) t

```

type (+'a,+'b) t =
| Empty
| Fork of ('b,'a) t * 'a * 'b * ('b,'a) t

let empty =
    Empty

```

The originality with this new tree type ('a,'b) t is that the left and right branches are not ('a,'b) t but ('b,'a) t. This is an irregular type recursion. Thus member, add and for_all operations will use polymorphic recursion plus will have a full explicit function type.

7.5. DoubleDimension, the member operation

It has exactly the same body code except it has full function type with 'a and 'b being further generalized as 'a 'b .

```

let rec member : 'a 'b . 'a -> 'b -> ('a,'b) t -> bool =
    fun x y -> function
    | Empty -> false
    | Fork (l,u,v,r) ->
        if x < u then member y x l
        else if x > u then member y x r
        else if y = v then true
        else member y x r

```

7.6. DoubleDimension, the add operation

It has exactly the same body code except it has full function type with 'a and 'b being further generalized as 'a 'b .

```

let rec add : 'a 'b . 'a -> 'b -> ('a,'b) t -> ('a,'b) t =
    fun x y -> function
    | Empty -> Fork(Empty,x,y,Empty)
    | Fork (l,u,v,r) as g ->
        if x < u then Fork (add y x l,u,v,r)
        else if x > u then Fork (l,u,v,add y x r)
        else if y = v then g
        else Fork (l,u,v,add y x r)

```


7.7. DoubleDimension, the first & second operations

The first operation is a new name for the old predecessors operation and the second operation is a new name for the old successors operation. Otherwise nothing has changed.

```
let apply f w acc = function
| Empty -> acc
| Fork (l,u,v,r) ->
    f w (f w (if w = v then u::acc else acc) r) l

let rec loop key acc = function
| Empty -> []
| Fork (l,u,v,r) ->
    if key < u then apply loop key acc l
    else if key > u then apply loop key acc r
    else v::apply loop key acc r

let first y =
    apply loop y []

let second x =
    loop x []
```

7.8. DoubleDimension, the for_all operation

The for_all operation tells if all elements in the set satisfy a predicate. What is the type of this predicate ? Either it is 'a->'b->bool or it is 'b->'a->bool. Actually it depends if we are at an even depth level or an odd depth level. We need both, we call them p and q and swap them at each new depth level.

```
let rec for_all :
  'a 'b .
  ('a -> 'b -> bool) -> ('b -> 'a -> bool) -> ('a,'b) t -> bool
=
  fun p q -> function
  | Empty -> true
  | Fork (l,u,v,r) -> for_all q p l && p u v && for_all q p r
```

Now we need to compute a 'b->'a->bool value from an 'a->'b->bool value.

```
(* apply 2 function arguments in swapped order *)
let flip f x y =
  f y x
```

Now we can define for_all in its simplest form.

```
(* checks if all elements satisfy the predicate p *)
let for_all p =
  for_all p (flip p)
```

7.9. DoubleDimension, the subset & equal operations

The subset operation is a new name for the old subgraph operation. It is now simply defined using `for_all`.

```
(* is ta a subset of tb ? *)
let subset ta tb =
  for_all (fun a b -> member a b tb) ta
```

Set equality is reciprocal inclusion :

```
(* set equality *)
let equal ta tb =
  subset ta tb && subset tb ta
```

7.10. DoubleDimension, the to_list operation

The `to_list t` operation converts `t` to a list of pairs. What is the type of these pairs ? Either it is `'a * 'b` or it is `'b * 'a`. Actually it depends if we are at an even depth level or an odd depth level. We call this new type `'c`. We need two functions to create a `'c` element from `'a` and `'b` elements, we call them `fu` and `fv` and swap them at each new depth level.

```
(* linearization *)
let rec to_list :
  'a 'b .
  'c list -> ('a->'b->'c) -> ('b->'a ->'c) -> ('a,'b) t -> 'c list
=
  fun acc fu fv -> function
  | Empty ->
    acc
  | Fork (l,u,v,r) ->
    to_list (fu u v::to_list acc fv fu r) fv fu l

let to_list t =
  to_list [] (fun u v -> u,v) (fun v u -> u,v) t
```

7.11. The RandomAccessList module

The `RandomAccessList` module is an alternative to the `BraunStack` module. It has the same operations but this time it stores polymorph `'a` values instead of

the monomorph int values. Another difference is that BraunStack is based on a Braun tree whereas RandomAccessList is based on base 2 numerals.

```
module RandomAccessList
  :
sig
  type +'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val singleton : 'a -> 'a t
  val size : 'a t -> int
  val add : 'a -> 'a t -> 'a t
  val member : int -> 'a t -> 'a
  val remove : 'a t -> 'a t
  val replace : int -> 'a -> 'a t -> 'a t
  val meld : 'a t -> 'a t -> 'a t
end
=
struct (*...*)
```

7.12. RandomAccessList, the new type 'a t

```
type +'a t =
  | Empty
  | Zero of ('a * 'a) t
  | One of 'a * ('a * 'a) t
```

The originality with this new list type 'a t is that the tail is not 'a t but ('a * 'a) t. This is an irregular type recursion. This time all operations will use polymorphic recursion plus will have a full explicit function type. Remark that One has both a head and a tail whereas Zero only has a tail. A Zero holds zero items whereas a One holds 2^n items.

```
let empty =
  Empty

let is_empty l =
  l = Empty
```

A singleton contains only one item.

```
let singleton x =
  One(x, Empty)
```

7.13. RandomAccessList, the size operation

The size operation is even faster than its BraunStack counterpart.

```
let rec size : 'a . 'a t -> int =
  function
  | Empty -> 0
  | Zero t -> 2 * size t
  | One (_,t) -> 1 + 2 * size t
```

7.14. RandomAccessList, the add operation

The add operation acts much like a base 2 number increment.

```
let rec add : 'a . 'a -> 'a t -> 'a t =
  fun x -> function
  | Empty -> One (x,Empty)
  | Zero t -> One(x,t)
  | One(y,t) -> Zero(add (x,y) t)
```

7.15. RandomAccessList, the member operation

Now we want to know the last added item or any older item.

The last item will be member 0, the last but 1 will be member 1, and so on ...

```
let rec member : 'a . int -> 'a t -> 'a =
  fun i -> function
  | Empty -> invalid_arg "RandomAccessList.member"
  | One (x,_) when i = 0 -> x
  | One (_,t) -> member (i - 1) (Zero t)
  | Zero t ->
    let x,y = member (i / 2) t in
    if i mod 2 = 0 then x else y
```

7.16. RandomAccessList, the remove operation

The remove operation acts much like a base 2 number decrement.

```
let rec remove : 'a . 'a t -> 'a * 'a t =
  function
  | Empty -> invalid_arg "RandomAccessList.remove"
  | One (x, Empty) -> x,Empty
  | One (x, t) -> x,Zero t
  | Zero t -> let (x,y),s = remove t in x,One(y,s)

let remove t =
  snd (remove t)
```

We keep only the tail part of the remove t operation because the head part is member 0 and we want consistency with the BraunStack.remove operation.

7.17. RandomAccessList, the replace operation

The replace `i x` operation binds the member `i` item to the value `x`.

```
let replace i v =
  let rec go : 'a . ('a -> 'a) -> int -> 'a t -> 'a t =
    fun f n -> function
      | Empty -> invalid_arg "RandomAccessList.replace"
      | One(x,t) ->
          if n=0 then One(f x,t)
          else add x (go f (n - 1) (Zero t))
      | Zero t ->
          let g (x,y) = if n mod 2 = 0 then (f x, y) else (x, f y)
          in Zero (go g (n / 2) t)
  in go (fun x -> v) i
```

7.18. RandomAccessList, the meld operation

The meld operation acts much like a base 2 number addition.

```
(* like an append but does not preserve item rank *)
let rec meld : 'a . 'a t -> 'a t -> 'a t =
  fun la lb -> match la,lb with
  | Empty      , ta          -> ta
  |      ta , Empty          -> ta
  | Zero  ta , Zero  tb  -> Zero (meld ta tb)
  | Zero  ta , One  (x, tb)
  | One  (x, ta), Zero  tb  -> One (x, meld ta tb)
  | One  (x, ta), One  (y, tb) -> Zero (add (x, y) (meld ta tb))
```

Chapter 8.

An adventure in data collections

8.1. The module language

We add the two keywords `include` and `with` to the module language. Then we challenge the expressive power of the module language to capture the elusive notion of a polymorph data collection.

8.2. The categorical functors

A categorical functor is a mapping from a category to a category of the same structure. Fortunately we don't even need to know what is a category, instead the source category will be named `'a t`, the target category will be named `'b t`, and the mapping will be named `map`.

```
module type Functor =  
sig  
  type 'a t  
  val map : ('a -> 'b) -> 'a t -> 'b t  
end
```

It's a tiny but familiar signature, much like `List` and `Array`.

8.3. The data collection skeleton

We extend the `Functor` module-type by adding the `size` and the `member` operations. What we obtain is a wanna-be minimalist notion of a polymorph data collection.

```
module type DataCollection =  
sig  
  include Functor  
  val size : 'a t -> int  
  type index  
  val member : index -> 'a t -> 'a  
end
```

Note that thanks to `include Functor` the module-type `DataCollection` is also a `Functor` module-type.

Now we will try and test our initial `DataCollection` intuition by (re)implementing polymorph data collections of various kinds.

8.4. The Stdlib collections

Because `'a t` has no variance we can implement both immutable collections like `Stdlib.List` and mutable collections like `Stdlib.Array`. We realize the abstract types `'a t` and `index` by using `include ... with type ... = ...` clauses.

```
(* Stdlib.List *)
module StdList
:
sig
  include DataCollection
  with type 'a t = 'a list
  with type index = int
end
=
struct
  type 'a t =
    'a list
  let map =
    List.map
  let size =
    List.length
  type index =
    int
  let member n l =
    List.nth l n
end

(* Stdlib.Array *)
module StdArray
:
sig
  include DataCollection
  with type 'a t = 'a array
  with type index = int
end
=
struct
  type 'a t =
    'a array
  let map =
    Array.map
  let size =
```

```

        Array.length
type index =
  int
let member n a =
  Array.get a n
end

```

8.5. The Pair collection

The Stdlib does not have a Pair module, we can remedy the omission.

```

module PairIndex
=
struct
  type index = Fst | Snd
end

(* a pair of values *)
module Pair
:
sig
  include DataCollection
  with type 'a t = 'a * 'a
  with type index = PairIndex.index
end
=
struct
  type 'a t =
    'a * 'a
  let map f (x,y) =
    (f x,f y)
  let size t =
    2
  type index =
    PairIndex.index
  let member i (x,y) =
    let open PairIndex in
    match i with
    | Fst -> x
    | Snd -> y
end

```

8.6. The Braun min-heap collection

We can also implement a full-fledged classic collection with information-hiding by keeping type 'a t abstract. We equip it with empty, add and the more

heap-specific operations replace, remove. The heap member operation always returns the minimum item hence the type `index = unit`.

```
module BraunHeap
:
sig
  include DataCollection
  with type index = unit
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val replace : 'a -> 'a t -> 'a t
  val remove : 'a t -> 'a t
end
=
struct
  type 'a t =
    | Empty
    | Fork of 'a t * 'a * 'a t
  let rec map f = function
    | Empty -> Empty
    | Fork(l,n,r) -> Fork(map f l,f n,map f r)
  let rec diff n = function
    | Empty -> if n = 0 then 0 else assert false
    | Fork(l,_,r) ->
      if n = 0 then 1
      else if n mod 2 = 1 then diff ((n - 1) / 2) l
      else diff ((n - 2) / 2) r
  let rec size = function
    | Empty -> 0
    | Fork(l,_,r) -> let m = size r in 2 * m + 1 + diff m l
  type index =
    unit
  let member () = function
    | Empty -> invalid_arg "BraunHeap.member"
    | Fork(_,n,_) -> n
  let empty =
    Empty
  let rec add n = function
    | Empty -> Fork(Empty,n,Empty)
    | Fork(l,m,r) ->
      if n < m then Fork(add m r,n,l)
      else Fork(add n r,m,l)
  let rec replace n = function
    | Empty -> invalid_arg "BraunHeap.replace"
    | Fork((Fork(_,m,_) as l),_,Empty)
      when m < n ->
      Fork(replace n l,m,Empty)
    | Fork((Fork(_,na,_) as l),_,(Fork(_,nb,_) as r))
      when na < n || nb < n ->
      if na < nb
      then Fork(replace n l,na,r)
```

```

        else Fork(l,nb,replace n r)
    | Fork(l,_,r) -> Fork(l,n,r)
let rec remove = function
| Empty -> invalid_arg "BraunHeap.remove"
| Fork(t,_,Empty) -> t
| Fork(Empty,_,_) -> assert false
| Fork((Fork (_,na,_) as l),_,(Fork( _,nb,_) as r)) ->
    if na < nb
    then Fork(r,na,remove l)
    else Fork(replace na r,nb,remove l)
end

```

8.7. The Nest collection

A nested datatype can also be viewed as a data collection.
 We use the `Pair.map` operation in our `Nest.map` operation.

```

module NestT
=
struct
    type 'a t =
        | Nil
        | Cons of 'a * ('a * 'a) t
end

(* a nested datatype from the paper "Nested Datatypes"
 * by Richard Bird and Lambert Meertens 1998
 *)
module Nest
:
sig
    include DataCollection
    with type 'a t = 'a NestT.t
    with type index = int
end
=
struct
    type 'a t =
        'a NestT.t
    let rec map : 'a 'b . ('a -> 'b) -> 'a t -> 'b t =
        fun f -> function
        | Nil -> Nil
        | Cons(h,t) -> Cons(f h,map (Pair.map f) t)
    let rec size : 'a . 'a t -> int = function
        | Nil -> 0
        | Cons(_,t) -> 1 + 2 * size t
    type index =
        int
    let rec member : 'a . index -> 'a t -> 'a =
        fun n -> function

```

```
| Nil -> failwith "Nest.member"
| Cons(h,t) ->
    if n=0 then h else
    let x,y = member (n/2) t in
    if n mod 2 = 0 then x else y
end
```

8.8. Discussion

How many collections are escaping our skeleton ? Not that many. Firstly, the binary search tree set has an unusual member : 'a -> 'a t -> bool deconstructor. Secondly, the collections based on a bi-functor (such as ('a, 'b) DoubleDimension.t) are a whole new story. However hard we try we will never completely capture the essence of data collections. Nevertheless the module language has proved its merits as a solid modeling tool.

Chapter 9.

Records and references

9.1. Record types

As we have already seen, combinations of sum and product types can create powerful and expressive abstractions. However, a product type with too many fields can easily become unwieldy. It may also be difficult to remember what every item means if they are the same type.

This is where records come in handy as a tuple alternative. They have named fields, so it's easy to convey the meaning of every field.

9.2. Record type declaration

A new record type can be declared using a pair of braces embracing a list of `field : type` separated by a semi-colon.

```
type vector = {x : float; y : float}
```

Of course you can declare mutually recursive record types.

```
type ta = {b:b}  
and tb = {a:a}
```

Or any mutually recursive record/algebraic types.

9.3. Record value construction and deconstruction

A new record value (or pattern) can be created using a pair of braces embracing a list of `field = value` separated by a semi-colon. When you create a new record value, all fields must be defined. Trying to create a partially defined record will cause a compilation error.

```
# let r = {x = 0.; y = 1.};;  
val r : vector = {x = 0.; y = 1.}
```

A mutually recursive record value can be created using a `let rec ... = {...}` and `... = {...}`

```
# let rec a = {b=b}
  and b = {a=a};;
val a : ta = {b = {a = <cycle>}}
val b : tb = {a = {b = <cycle>}}
```

Record fields can be read using a dot notation.

```
# r.x +. r.y;;
- : float = 1.
```

9.4. Record field punning

When the name of a variable coincides with the name of a record field you are allowed to omit the `field=` bit.

```
# let rec a = {b}
  and b = {a};;
```

9.5. The queue FIFO datatype

What we know about records suffice to implement a First-In-First-Out queue abstraction.

```
module Queue
:
sig
  type +'a t
  val empty : 'a t
  val member : 'a t -> 'a
  val add : 'a -> 'a t -> 'a t
  val remove : 'a t -> 'a t
end
```

We could implement it using a simple list. Then the dilemma is that a member-to-add list has a linear-time add operation but a constant-time member operation whereas a reverse-ordered list has a constant-time add operation but a linear-time member operation. A solution is to have both a front list and a reversed rear list.

Firstly we declare the type `'a t` and define the empty value and the constant-time member operation.

```
type +'a t =
{front:'a list;rear:'a list}
let empty =
{front=[];rear=[]}
```

```
let member {front;_} =
  match front with
  | [] -> failwith "Queue.member"
  | h::_ -> h
```

Secondly, we define a queue smart constructor that will enforce the datatype invariant : a queue with an empty front can't have a non-empty rear. This constructor will eventually migrate items from rear to front.

```
let queue front rear =
  match front with
  | [] -> {front=List.rev rear;rear=[]}
  | _ -> {front;rear}
```

Thirdly, we define the remaining add and remove operations using only our queue constructor. As a result the duo add-remove operates in amortized-constant-time.

```
let add n {front;rear} =
  queue front (n::rear)
let remove {front;rear} =
  match front with
  | [] -> failwith "Queue.remove"
  | _::t -> queue t rear
```

9.6. Inline records

An anonymous record can be used instead of a tuple in an algebraic datatype.

```
type +'a t =
  | Empty
  | Fork of {left:'a t;item:'a;right:'a t}
```

9.7. Immutable records and functional update

Records have a copy & modify operation using the with keyword.

```
let rec add x t =
  match t with
  | Empty -> Fork {left=Empty;item=x;right=Empty}
  | Fork n ->
    if x < n.item then Fork {n with left=add x n.left}
    else if x > n.item then Fork {n with right=add x n.right}
    else t
```

9.8. Mutable fields and destructive update

Instead of a functional update you can do a destructive update by using the `mutable` keyword and the `<-` symbol.

```
type 'a t =
  | Empty
  | Fork of {mutable left:'a t;item:'a;mutable right:'a t}
let rec add x t =
  match t with
  | Empty -> Fork {left=Empty;item=x;right=Empty}
  | Fork n ->
    if x < n.item then (n.left <- add x n.left; t)
    else if x > n.item then (n.right <- add x n.right; t)
    else t
```

Except the `'a t` type variance there is little difference between the functional style and the destructive style.

9.9. The garbage collector write-barrier

There is little difference in the code however there may be a bigger difference in the performance. Due to the efficient compacting garbage collector design, every destructive update issues a call to the `caml_modify` procedure. As a result you have to resort to benchmarking to (in)validate your intuition that the destructive style is faster.

9.10. References

Creating a new record type every time you need a single mutable variable would quickly become cumbersome, so OCaml standard library includes very simple syntactic sugar for that case.

There's a type `'a ref` that is really a record with a single mutable field named `contents`. It comes with a function `ref : 'a -> 'a ref` that constructs a new reference, and an assignment operator `:=`. There's also a prefix deconstructor `!` for getting the content value.

```
let fibonacci n =
  let a = ref 1 and b = ref 1 and c = ref 2 in
  while !n > 1 do
    c := !a + !b; a := !b; b := !c;
    decr n;
  done;
  !b
```

9.11. Record rank-2 polymorphism

If you are familiar with the lambda-calculus you may be tempted to provide an arithmetic module using a Church-encoding of natural numbers. It is fairly straightforward and you end up with a polymorphic ChurchArith.t type. Somehow, every number is a number of something.

```

module ChurchArith
:
sig
  type 'a t
  val zero : 'a t
  val one : 'a t
  val two : 'a t
  val three : 'a t
  val inc : 'a t -> 'a t
  val add : 'a t -> 'a t -> 'a t
  val mul : 'a t -> 'a t -> 'a t
  val power : 'a t -> ('a -> 'a) t -> 'a t
  val to_int : int t -> int
end
=
struct
  type 'a t =
    ('a -> 'a) -> ('a -> 'a)
  let zero f x =
    x
  let one f x =
    f x
  let two f x =
    f (f x)
  let three f x =
    f (f (f x))
  let inc n f x =
    f (n f x)
  let add n m f x =
    n f (m f x)
  let mul n m f x =
    n (m f) x
  let power n m f x =
    m n f x
  let to_int t =
    t succ 0
end

module CA = ChurchArith

# CA.to_int (CA.power CA.three CA.two);;
- : int = 9

```

Everything looks fine, much like Coq nat arithmetic. Until we deal with ChurchArith.t itself :


```
# CA.add CA.three CA.two;;
- : '_weak1 CA.t = <abstr>
```

What an horror, weak polymorphism strikes again ! Clearly the ChurchArith.t type is too polymorphic for our own good. How to make it monomorphic ? One solution is to restrict the scope of the 'a universal quantification. What we want to be universally quantified is the ('a -> 'a) -> ('a -> 'a) type, not the ChurchArith.t type itself. An appropriate record declaration creates a new scope where the 'a is migrated.

```
module ChurchArith
:
sig
  type t
  val zero : t
  val one : t
  val two : t
  val three : t
  val inc : t -> t
  val add : t -> t -> t
  val mul : t -> t -> t
  val power : t -> t -> t
  val to_int : t -> int
end
=
struct
  type t =
    {c: 'a. ('a -> 'a) -> ('a -> 'a)}
  let zero =
    {c = fun f x -> x}
  let one =
    {c = fun f x -> f x}
  let two =
    {c = fun f x -> f (f x)}
  let three =
    {c = fun f x -> f (f (f x))}
  let inc n =
    {c = fun f x -> f (n.c f x)}
  let add n m =
    {c = fun f x -> n.c f (m.c f x)}
  let mul n m =
    {c = fun f x -> n.c (m.c f) x}
  let power n m =
    {c = fun f x -> m.c n.c f x}
  let to_int t =
    t.c succ 0
end

module CA = ChurchArith
```

```
# CA.to_int (CA.power CA.three CA.two);;  
- : int = 9  
# CA.add CA.three CA.two;;  
- : CA.t = <abstr>
```

Yes, the code beauty has faded a little, but at least the monomorphic / strongly polymorphic duopole is restored.

Chapter 10.

Modules as functions

10.1. The polymorphic SquareMatrix module

Using type polymorphism as seen in chapter 6 we can quickly design a module for square matrices equipped with an addition operation.

```
module SquareMatrix
:
sig
  type 'a t
  val make : int -> 'a -> 'a t
  val add : ('a -> 'a -> 'a ) -> 'a t -> 'a t -> 'a t
end
=
struct
  type 'a t =
    'a array array
  let make size x =
    Array.make_matrix size size x
  let add plus ma mb =
    Array.map2 (Array.map2 plus) ma mb
end
```

Although this SquareMatrix module is effective its add operation is somewhat fragile :

- size inequality is only detected at run-time. We would prefer it to be detected at compile-time.
- there is no guarantee that the same plus operator is consistently used. Again we would prefer it to be fixed at compile-time.

10.2. The monomorphic SquareMatrix module-function

A module-function is a function from a module-type to another module-type. The argument module-type contains all what we want to be fixed at compile-time (size and plus in our case). The argument module(-type) is always parenthesized.

```
module SquareMatrix
```

```

(M :
sig
  type t
  val size : int
  val plus : t -> t -> t
end
)
:
sig
  type t
  val make : M.t -> t
  val add : t -> t -> t
end
=
struct
  type t =
    M.t array array
  let make x =
    Array.make_matrix M.size M.size x
  let add ma mb =
    Array.map2 (Array.map2 M.plus) ma mb
end

```

The `Stdlib.Set.Make` and `Stdlib.Map.Make` module-functions use a similar design to enforce a consistent usage of the compare operator.

10.3. Using the parameterized `SquareMatrix` module

Since `SquareMatrix` is now monomorphic, before use (of make/add operations) each module instance must be given a unique name and be created by applying the wanted module-argument to the module-function.

```

module IntMatrix2x2 =
  SquareMatrix(struct type t=int let size=2 let plus=(+) end)
module FloatMatrix3x3 =
  SquareMatrix(struct type t=float let size=3 let plus=(+.) end)
module IntMatrixMatrix2x2 =
  SquareMatrix
    (struct type t=IntMatrix2x2.t let size=2 let plus=IntMatrix2x2.add
    end)

```

10.4. Applicative module-functions

OCaml module-functions are applicative by default, that means given these 3 modules :

```

module Int2 = struct type t=int let size=2 let plus=(+) end
module IntMatrix2x2 = SquareMatrix(Int2)
module IntMatrix2x2bis = SquareMatrix(Int2)

```

IntMatrix2x2 and IntMatrix2x2bis are the same module-function and IntMatrix2x2.t and IntMatrix2x2bis.t are the same type.

10.5. Generative module-functions

On the contrary, sometimes you want to make some type unique. It's made possible by using an empty module-argument. As a use-case you often need float computations to be restricted to certain domains. Especially you don't want to mix different measures like meters and kilograms or dollars and euros. All you have to do is packing a float type `t` and all needed operators in a generative Measure module.

```
module Measure ()
:
sig
  type t
  val add: t -> t -> t
  (* other needed operators *)
end
=
struct
  type t = float
  let add = (+.)
  (* other needed operators *)
end;;

module Meter = Measure ();;
module Kilogram = Measure ();;
module Dollar = Measure ();;
module Euro = Measure ();;
```

Now `Meter.t`, `Kilogram.t`, `Dollar.t` and `Euro.t` are 4 different types that can't be inadvertently mixed.

Another use-case is to make some value(s) unique. That makes the generative module much like a glorified record.

```
module Counter () =
struct
  let ticks = ref 0
  let tick () = incr ticks
  let count () = !ticks
end
```

Now each `Counter ()` has a unique `count ()` value.

```
# module CA = Counter ();;
module CA :
  sig val ticks : int ref val count : unit -> int val tick : unit ->
    unit end
# CA.tick ();;
- : unit = ()
# CA.count ();;
- : int = 1
# module CB = Counter ();;
module CB :
  sig val ticks : int ref val count : unit -> int val tick : unit ->
    unit end
# CB.count ();;
- : int = 0
```

Chapter 11.

ERic v0.3 & Conceptual graphs

11.1. Abstract hyper-graphs

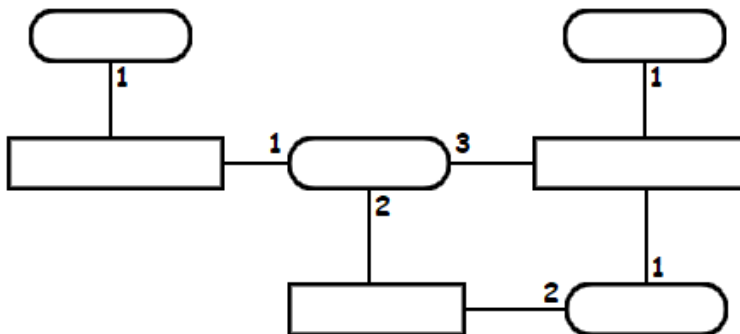
An abstract hyper-graph $G(V,E)$ consists of a set V of vertices and a multiset E of hyper-edges.

Every hyper-edge in E is a tuple (v_1, v_2, \dots, v_N) of vertices from the set V .

Thus an hyper-edge can connect many (1 to n) vertices.

There are graphical conventions to present an abstract hyper-graph. A vertex is drawn as an empty square box whereas an hyper-edge is drawn as an empty rounded box with each connection labeled with number 1 to n .

Example.

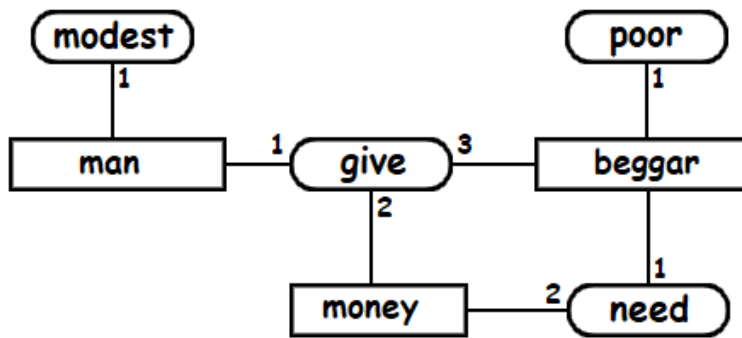


11.2. Concrete hyper-graphs

A concrete hyper-graph $G(V,E,L_V,L_E)$ is an abstract hyper-graph $G(V,E)$ where each vertex is labeled with an element of L_V and each hyper-edge is labeled with an element of L_E .

Example.

A modest man gives some money to a poor beggar in need.

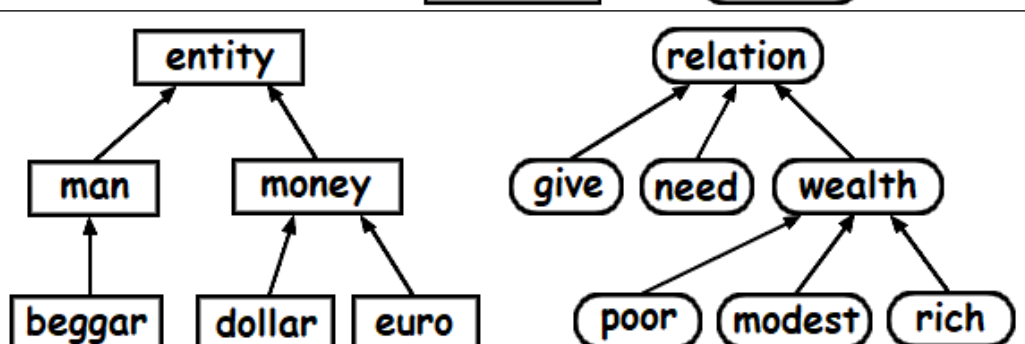
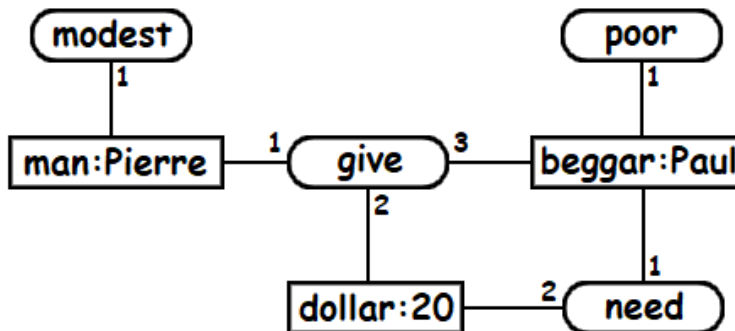


11.3. Conceptual graphs

A conceptual-graph $G(V,E,H_V,L_V,H_E)$ is an abstract hyper-graph $G(V,E)$ where :

- Each hyper-edge is labeled with an element of H_E where H_E is a hierarchy of hyper-edge labels.
- Each vertex is labeled with an element of H_V where H_V is a hierarchy of vertex labels and eventually also labeled with an element of L_V where L_V is a set of vertex labels.

Example : Pierre, a modest man, gives 20 dollars to Paul, a poor beggar in need.



entity is the top-most concept of the H_V hierarchy.
relation is the top-most relation of the H_E hierarchy.

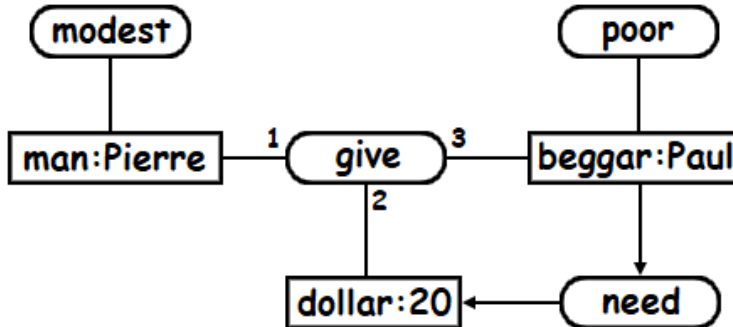
Pierre, Paul, 20 are elements of the set L_V .

11.4. More graphical conventions

In order to streamline the graphical notation :

- an edge connected to only 1 vertex is exempted from the 1 label.
- an edge connected to 2 vertices will have an input arrow & an output arrow instead of 1 & 2 labels.
- thus only ternary (and more) hyper-edges will retain connection labels.

Example : Pierre, a modest man, gives 20 dollars to Paul, a poor beggar in need.

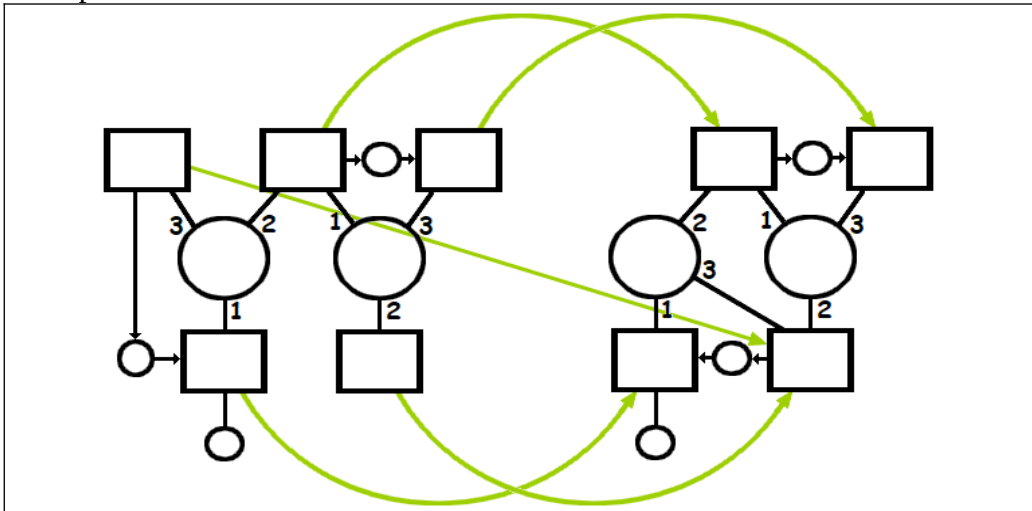


11.5. Abstract hyper-graph homomorphisms

An abstract hyper-graph homomorphism H from an abstract hyper-graph $G(V,E)$ to an abstract hyper-graph $G'(V',E')$ is a mapping h from V to V' such that :

- informally : h preserves the adjacency relation.
- formally : if $(v_1, v_2, \dots, v_N) \in E$ then $(h(v_1), h(v_2), \dots, h(v_N)) \in E'$.

Example :



The abstract-hyper-graphs $G(V,E)$ and $G'(V',E')$.
 The green arrows is a mapping from V to V' .

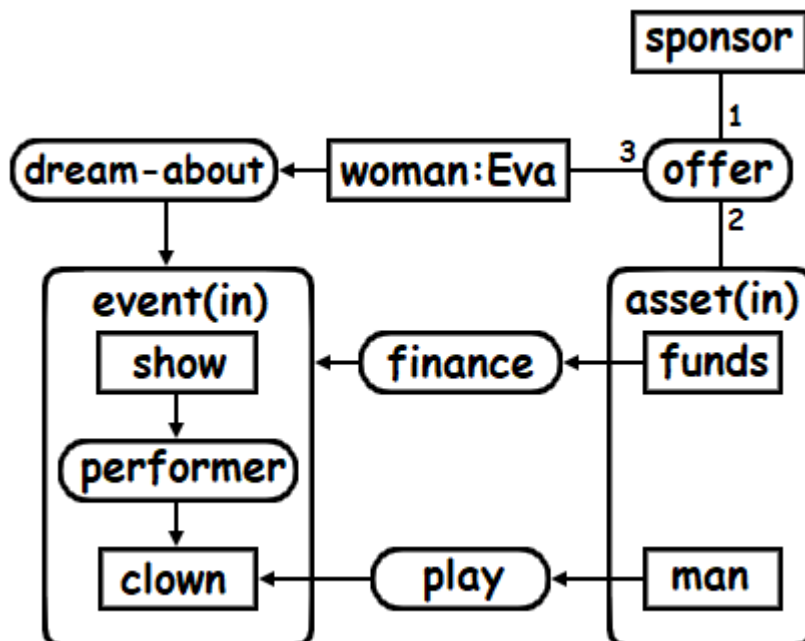
11.6. Conceptual graph homomorphisms and subsumption

The concept / relation hierarchies explicit an is-a relation between lower concepts / relations and higher concepts / relations. Similarly there is also an is-a relation between an entity and its concept. And finally there is an is-a relation between an abstract hyper-graph G' and another hyper-graph G that maps to G' by an homomorphism. When the three are taken together there is an is-a relation between G' and G . Then it is said that the conceptual-graph G subsumes the conceptual-graph G' .

11.7. Nested conceptual graphs

A nested conceptual graph is a conceptual graph where one (or more) square box contains further conceptual graph(s). Eventually, nested conceptual graphs allow edges to traverse square box boundaries in any manner (from inside to outside, from outside to inside, whatever the nesting level). Nested graphs provide a direct visual way to decorate the information with the associated context.

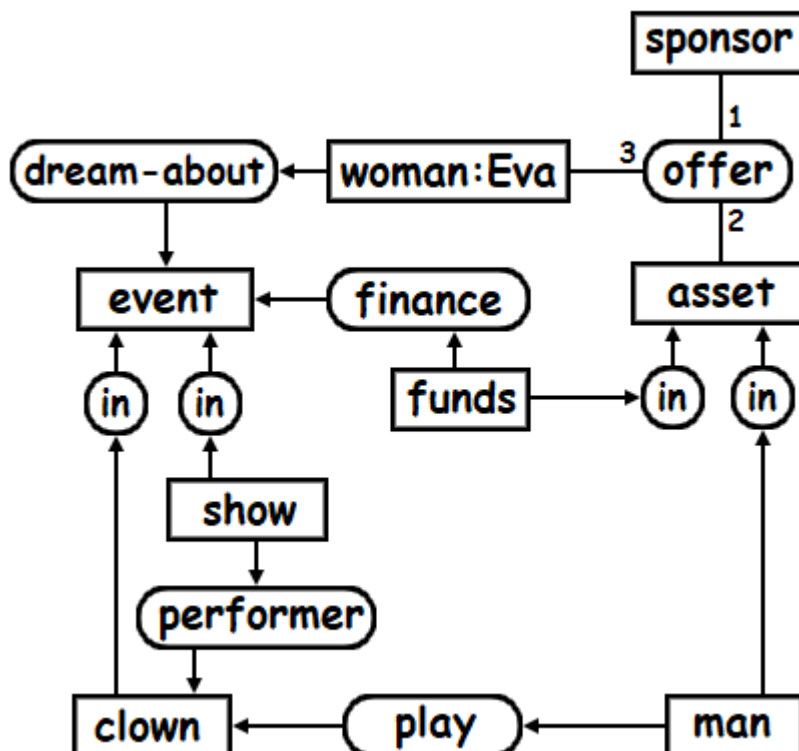
Example. Eva dreams about a clown show. A sponsor makes her an offer of a man to act as the clown and the funds to finance the event.



A nested conceptual graph can be unfolded, that is no box is nested, instead inner boxes are connected by a relation to outer boxes. In our example nesting is a convenient way to distribute the (in) relation.

Unfolded example.

Eva dreams about a clown show. A sponsor makes her an offer of a man to act as the clown and the funds to finance the event.



11.8. Nested conceptual graphs and subsumption

Theorem : there exists a bijection between unfolded conceptual graphs and nested conceptual graphs. As a consequence the subsumption between nested conceptual graphs is the same as subsumption between unfolded conceptual graphs.

11.9. CGIF (Conceptual Graph Interchange Format) notation

CGIF is a normalized textual notation for conceptual graphs communication. ERic v0.3 uses a CGIF dialect instead of the more appealing visual notation.

As a first demonstration our first conceptual graph translates to the following verbatim :

```
insert
[man:Pierre]
[beggar:Paul]
(modest Pierre)
(poor Paul)
(give Pierre [dollar:20] Paul)
(need Paul [dollar:20]).
```

As a second demonstration our second conceptual graph translates to the following verbatim :

```
insert
[woman:Eva]
(offer [sponsor:*] [asset:*a] Eva)
(dream-about Eva [event:*e])
(performer [show:*s] [clown:*c])
(in ?s ?e)
(in ?c ?e)
(finance [funds:*f] ?e)
(play [man:*m] ?c)
(in ?f ?a)
(in ?m ?a).
```

Of course both demonstrations build upon concept / relation hierarchies. In the first case the hierarchies verbatim could be :

```
untyped hierarchy entity relation.
```

```
derive entity man money.
derive man beggar.
derive money dollar euro yen.
```

```
derive1 relation1 wealth.
derive2 relation2 need.
derive3 relation3 give.
derive1 wealth poor modest rich.
```

In the second case the hierarchies verbatim could be :

```
untyped hierarchy entity relation.
```

```
derive entity man woman sponsor asset event clown funds show.
derive2 relation2 in dream-about performer finance play.
derive3 relation3 offer.
```

A much more substantial example is this command line:

```
./ERic <anatomy-of-a-toy.gif
```

11.10. Other ERic commands

select *cgif-graph*. Find all is-a homomorphisms from the *cgif-graph* to the knowledge database.
quit. Quit the program.
save "*filename*". Save the database to the binary *filename*.
load "*filename*". Erase all and load the database from the binary *filename*.
derive4 *relation4 relations*. Add new quaternary *relations* to the dictionary.

Along with signed integers ERic accepts signed floats and string constants.

11.11. ERic community & forum

The [official ERic community & forum](http://www.developpez.net) is kindly hosted at www.developpez.net

The [official ERic SVN repository](#).

11.12. Going further

ERic v0.3 is merely a Knowledge Representation toy written in less than 1300 lines to demonstrate the [OCaml programming language](#). If you need a full-fledged KR tool i recommend you to install [CoGui 3.1](#) by the GraphiK team at LIRMM, Montpellier.

Chapter .

The art of searching

1. Searching

An alert reader of the "art of sorting" chapter may ask what do programmers when not sorting ? A probable appropriate answer could be : they are searching. Searching what ? Searching solutions for a CSP ([Constraint Satisfaction Problem](#)). OCaml type inference is an example of a sophisticated CSP. If you have read the chapter 11 you certainly have anticipated that our CSP concern is finding subsumed hyper-graph-homomorphisms. Nonetheless the techniques presented here apply to all kinds of CSP. Another thing to keep in mind is that finding **all** solutions can be more important than finding **one** (even one good enough) solution.

2. Brute force searching

The most naïve approach is to enumerate all possible positions and filter them according to the problem constraint. This approach is typically un-practicable but to a tiny problem size.

3. Chronological backtracking

The chronological backtracking approach is to incrementally-decrementally build partial legal positions. When a valid partial position is reached then move forward. When an invalid partial position is reached then move backward. Continue until all total valid positions are reached. This approach is typically illustrated using a chess problem.

4. The Knight Tour problem backtracking

Put a knight on a chessboard corner square. The knight tour problem is to jump from square to square until all chessboard squares have been visited exactly once.

```
let board =  
    Array.make_matrix 8 8 0
```

```

let bounded x y =
  if x < 0 || x > 7 then false
  else if y < 0 || y > 7 then false
  else true

let print_board () =
  Array.iter (fun a -> Array.iter
    (fun k -> if k < 10 then print_char '0'; print_int k;
print_char ' ') a;
    print_newline ())
    board

let rec forward_chaining n x y =
  if bounded x y && board.(x).(y) = 0 then begin
    board.(x).(y) <- n;
    if n = 8 * 8 then
      (print_board (); print_newline ())
    else begin
      forward_chaining (n + 1) (x + 1) (y + 2);
      forward_chaining (n + 1) (x + 2) (y + 1);
      forward_chaining (n + 1) (x + 2) (y - 1);
      forward_chaining (n + 1) (x + 1) (y - 2);
      forward_chaining (n + 1) (x - 1) (y - 2);
      forward_chaining (n + 1) (x - 2) (y - 1);
      forward_chaining (n + 1) (x - 2) (y + 1);
      forward_chaining (n + 1) (x - 1) (y + 2);
    end;
    board.(x).(y) <- 0;
  end

let () = forward_chaining 1 0 0

```

This program prints all Knight Tour solutions until you are fed up and press Ctrl + C. Note that `bounded x y && board.(x).(y)=0` is possible only because `&&` is semi-strict.

5. The 8-Queens problem backtracking

The 8-queens problem is to place 8 queens on a regular chessboard without any threatening possible.

The brute-force solution would be to examine all permutations of `[1;2;...;8]` and filter the ones satisfying the "no two diagonally threaten" constraint. At first the brute force approach is attractive : it's simple and fully functional. At a second sight it's less desirable because it does not scale up well. What if we generalize to N-Queens when N is much larger than 8 ? Then, for efficiency reason, we have to switch to backtracking. The program will be more procedural and much less elegant.

How do we proceed ? We go the object-oriented way. Each queen object has a constant column and a mutable row. Queen rows increase from 1 to 8 then reset to 1 again. When a successful partial solution is found then vertically move the next queen, otherwise return back and vertically move the previous queen. Finally, the first, next and print methods offer a nice dripping display of the solutions.

```

let null_queen =
  object
    method first = false
    method next = false
    method check (x:int) (y:int) = false
    method print = print_newline ()
  end

class queen column (neighbor:queen) =
  object (self)
    val mutable row = 1
    method check r c =
      let diff = c - column in
      (row = r) ||
      (row + diff = r) || (row - diff = r) ||
      neighbor#check r c
    method private test =
      try
        while neighbor#check row column do
          if not self#advance then raise Exit
        done;
        true
      with
        | Exit -> false
    method first =
      ignore(neighbor#first);
      row <- 1;
      self#test
    method next =
      self#advance && self#test
    method private advance =
      if row = 8 then
        (if neighbor#next then (row <- 1; true) else false)
      else (row <- row + 1; true)
    method print : unit =
      print_int row; print_char ' '; neighbor#print
  end

let () =
  let last = ref null_queen in
  for i = 1 to 8 do
    last := new queen i !last
  done;

```



```

if !last#first then begin
  !last#print;
  while !last#next do
    !last#print
  done
end else
  print_string "Zero solution."

```

This program prints all 8-Queens 92 solutions.

6. Brute force 8-Queens problem

For the sake of code functional beauty here is the brute force version of the 8-Queens problem.

```

let rec distribute a l =
  match l with
  | [] -> [[]]
  | h::t -> (a::l) :: List.map (fun x -> h::x) (distribute a t)

let rec permute = function
  | [] -> [[]]
  | h::t -> List.flatten (List.map (distribute h) (permute t))

let rec one_safe low high = function
  | [] -> true
  | h::t -> h <> low && h <> high && one_safe (low - 1) (high + 1) t

let rec all_safe = function
  | [] -> true
  | h::t -> one_safe (h - 1) (h + 1) t && all_safe t

let _ =
  List.filter all_safe (permute [1;2;3;4;5;6;7;8])

```

7. Backjumping