Curious OCaml

Lukasz Stafiniak

Contents

Curious OCaml

Chapter 1: Logic

From logic rules to programming constructs

1.1 In the Beginning there was Logos

What logical connectives do you know?

$\top$	$\bot$	$\wedge$	$\vee$	$\rightarrow$
		$a \wedge b$	$a \vee b$	$a \rightarrow b$
truth	falsehood	conjunction	disjunction	implication
"trivial"	"impossible"	a and b	a or b	a gives b
	shouldn't get	got both	got at least one	given a, we get b

How can we define them? Think in terms of *derivation trees*:

$$\dfrac{\dfrac{\overline{\text{a premise}} \quad \overline{\text{another premise}}}{\text{some fact}} \qquad \dfrac{\overline{\text{this we have by default}}}{\text{another fact}}}{\text{final conclusion}}$$

We define connectives by providing rules for using them. For example, a rule $\frac{a\ b}{c}$ matches parts of the tree that have two premises, represented by variables a and b, and have any conclusion, represented by variable c.

Design principle: Try to use only the connective you define in its definition.

1.2 Rules for Logical Connectives

Introduction rules say how to *produce* a connective.

Elimination rules say how to *use* it.

Text in parentheses is comments. Letters are variables that can stand for anything.

Connective	Introduction Rules	Elimination Rules
$\top$	$\frac{}{\top}$	doesn't have
$\bot$	doesn't have	$\frac{\bot}{a}$ (i.e., anything)
$\wedge$	$\frac{a \quad b}{a \wedge b}$	$\frac{a \wedge b}{a}$ (take first) $\frac{a \wedge b}{b}$ (take second)
$\vee$	$\frac{a}{a \vee b}$ (put first) $\frac{b}{a \vee b}$ (put second)	$\dfrac{a \vee b \qquad \overset{[a]^x}{\vdots}\ c \qquad \overset{[b]^y}{\vdots}\ c}{c}$ using x, y
$\rightarrow$	$\dfrac{\overset{[a]^x}{\vdots}\ b}{a \rightarrow b}$ using x	$\frac{a \rightarrow b \quad a}{b}$

Notation for Hypothetical Derivations The notation $\overset{[a]^x}{\vdots}\ b$ (sometimes written as a tree) matches any subtree that derives b and can use a as an assumption (marked with label x), even though a might not otherwise be warranted.

For example, we can derive "sunny → happy" by showing that *assuming* it's sunny, we can derive happiness:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\text{sunny}}^{\,x}}{\text{go outdoor}}}{\text{playing}}}{\text{happy}}}{\text{sunny} \to \text{happy}} \text{ using } x$$

Such assumptions can only be used in the matched subtree! But they can be used several times. For example, if someone's mood is more difficult to influence:

$$\dfrac{\dfrac{\dfrac{\overline{\text{sunny}}^{\,x}}{\text{go outdoor}}}{\text{playing}} \quad \dfrac{\overline{\text{sunny}}^{\,x}}{\text{nice view}} \quad \dfrac{\overline{\text{sunny}}^{\,x}}{\text{go outdoor}}}{\text{happy}}}{\text{sunny} \to \text{happy}} \text{ using } x$$

Reasoning by Cases The elimination rule for disjunction represents **reasoning by cases**.

How can we use the fact that it is sunny ∨ cloudy (but not rainy)?

$$\dfrac{\dfrac{}{\text{sunny} \vee \text{cloudy}}\,\text{forecast} \quad \dfrac{\overline{\text{sunny}}^{\,x}}{\text{no-umbrella}} \quad \dfrac{\overline{\text{cloudy}}^{\,y}}{\text{no-umbrella}}}{\text{no-umbrella}} \text{ using } x, y$$

We know that it will be sunny or cloudy (by watching the weather forecast). If it will be sunny, we won't need an umbrella. If it will be cloudy, we won't need an umbrella. Therefore, we won't need an umbrella.

Reasoning by Induction We need one more kind of rule to do serious math: **reasoning by induction** (somewhat similar to reasoning by cases). Example rule for induction on natural numbers:

$$\dfrac{p(0) \quad \begin{array}{c}[p(x)]^{x} \\ \vdots \\ p(x+1)\end{array}}{p(n)} \text{ by induction, using } x$$

We get property p for any natural number n, provided we can: 1. Establish $p(0)$ (the base case) 2. Show that assuming $p(x)$ holds, we can derive $p(x+1)$ (the inductive step)

Here x is a unique variable—we cannot substitute a particular number for it because we write "using x" on the side.

1.3 Logos was Programmed in OCaml

There is a deep correspondence between logic and programming, known as the **Curry-Howard correspondence** (or "propositions as types"). The following table shows how logical connectives correspond to programming constructs:

Logic	Type	Expression
$\top$	`unit`	`()`
$\bot$	`'a`	`raise`
$\wedge$	`*`	`(,)`
$\vee$	`|`	`match`
$\to$	`->`	`fun`
induction	—	`rec`

Typing rules for OCaml constructs:

- **Unit (truth):** $\frac{}{\texttt{()}:\texttt{unit}}$
- **Exception (falsehood):** $\frac{\text{oops!}}{\texttt{raise exn}:c}$ — can produce any type
- **Pair (conjunction):**
  - Introduction: $\frac{s:a \quad t:b}{(s,t):a*b}$
  - Elimination: $\frac{p:a*b}{\texttt{fst}\ p:a}$ and $\frac{p:a*b}{\texttt{snd}\ p:b}$
- **Variant (disjunction):**
  - Introduction: $\frac{s:a}{\texttt{A}(s):\texttt{A of}\ a\ \texttt{|}\ \texttt{B of}\ b}$
  - Elimination (match): given t of variant type and branches for each case, produce result c
- **Function (implication):**
  - Introduction: $\frac{\overset{[x:a]}{e:b}}{\texttt{fun}\ x\to e:a\to b}$
  - Elimination (application): $\frac{f:a\to b \quad t:a}{f\ t:b}$
- **Recursion (induction):** $\frac{\overset{[x:a]}{e:a}}{\texttt{rec}\ x=e:a}$

1.3.1 Definitions Writing out expressions and types repetitively is tedious: we need definitions.

Type definitions are written: `type ty =` some type.

- Writing `A(s) : A of a | B of b` in the table was cheating. Usually we have to define the type and then use it. For example, using `int` for a and `string` for b:

```
type int_string_choice = A of int | B of string
```

This allows us to write `A(s) : int_string_choice`.

- Without the type definition, it is difficult to know what other variants there are when one *infers* (i.e., "guesses", computes) the type!
- In OCaml we can write `` `A(s) : [`A of a | `B of b] ``. With "`" variants (polymorphic variants), OCaml does guess what other variants there are. These types are interesting, but we will not focus on them in this book.
- Tuple elements don't need labels because we always know at which position a tuple element stands. But having labels makes code more clear, so we can define a *record type*:

  ```
  type int_string_record = {a: int; b: string}
  ```

  and create its values: `{a = 7; b = "Mary"}`.
- We access the *fields* of records using the dot notation: `{a=7; b="Mary"}.b = "Mary"`.

1.3.2 Expression Definitions The recursive expression `rec x = e` in the table was cheating: `rec` (usually called `fix` in theory) cannot appear alone in OCaml! It must be part of a definition.

Definitions for expressions are introduced by rules a bit more complex:

$$\frac{e_1 : a \quad \frac{[x:a]}{e_2:b}}{\texttt{let } x = e_1 \texttt{ in } e_2 : b}$$

(Note that this rule is the same as introducing and eliminating $\rightarrow$.)

For recursive definitions:

$$\frac{\frac{[x:a]}{e_1:a} \quad \frac{[x:a]}{e_2:b}}{\texttt{let rec } x = e_1 \texttt{ in } e_2 : b}$$

We will cover what is missing in the above rules when we discuss **polymorphism**.

1.3.3 Scoping Rules

- **Type definitions** we have seen above are *global*: they need to be at the top-level (not nested in expressions), and they extend from the point they occur till the end of the source file or interactive session.
- **`let-in` definitions** for expressions: `let x = e1 in e2` are *local*—x is only visible in e_2. But **`let` definitions** without `in` are global: placing `let x = e1` at the top-level makes x visible from after e_1 till the end of the source file or interactive session.

- In the interactive session (toplevel/REPL), we mark the end of a top-level "sentence" with `;;`—this is unnecessary in source files.

1.3.4 Operators Operators like `+`, `*`, `<`, `=` are names of functions. Just like other names, you can use operator names for your own functions:

```
let (+:) a b = String.concat "" [a; b]  (* Special way of defining *)

"Alpha" +: "Beta"  (* but normal way of using operators *)
```

Operators in OCaml are **not overloaded**. This means that every type needs its own set of operators: `-` `+`, `*`, `/` work for integers - `+.`, `*.`, `/.` work for floating point numbers

Exception: Comparisons `<`, `=`, etc. work for all values other than functions.

1.4 Exercises

Exercises from *Think OCaml: How to Think Like a Computer Scientist* by Nicholas Monje and Allen Downey.

1. Assume that we execute the following assignment statements:

   ```
   let width = 17
   let height = 12.0
   let delimiter = '.'
   ```

   For each of the following expressions, write the value of the expression and the type (of the value of the expression), or the resulting type error.

   1. `width/2`
   2. `width/.2.0`
   3. `height/3`
   4. `1 + 2 * 5`
   5. `delimiter * 5`

2. Practice using the OCaml interpreter as a calculator:

   1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5? (*Hint:* 392.6 is wrong!)
   2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
   3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

3. You've probably heard of the Fibonacci numbers before, but in case you

haven't, they're defined by the following recursive relationship:

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(n+1) = f(n) + f(n-1) \quad \text{for } n = 2, 3, \ldots \end{cases}$$

Write a recursive function to calculate these numbers.

4. A palindrome is a word that is spelled the same backward and forward, like "noon" and "redivider". Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

   The following are functions that take a string argument and return the first, last, and middle letters:

```
let first_char word = word.[0]
let last_char word =
  let len = String.length word - 1 in
  word.[len]
let middle word =
  let len = String.length word - 2 in
  String.sub word 1 len
```

   1. Enter these functions into the toplevel and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string `""`?
   2. Write a function called `is_palindrome` that takes a string argument and returns `true` if it is a palindrome and `false` otherwise.

5. The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

   One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b, then $\gcd(a, b) = \gcd(b, r)$. As a base case, we can consider $\gcd(a, 0) = a$.

   Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor.

   If you need help, see http://en.wikipedia.org/wiki/Euclidean_algorithm.

Chapter 2: Algebra

Algebraic Data Types and some curious analogies

2.1 A Glimpse at Type Inference

For a refresher, let us apply the type inference rules introduced in Chapter 1 to some simple examples. We will start with the identity function `fun x -> x`. In the derivations below, [?] means "type unknown yet."

We begin with an incomplete derivation:

$$\frac{}{\texttt{fun x -> x} : [?]}\,[?]$$

Using the $\to$ introduction rule, we need to derive the body `x` assuming `x` has some type a:

$$\frac{\frac{}{\texttt{x}:a}\,x}{\texttt{fun x -> x} : [?] \to [?]}$$

The premise $\frac{}{\texttt{x}:a}\,x$ matches the pattern for hypothetical derivations since $e = \texttt{x}$. Since the body `x` has type a (from our assumption), and the parameter `x` also has type a, we conclude:

$$\frac{\frac{}{\texttt{x}:a}\,x}{\texttt{fun x -> x} : a \to a}$$

Because a is arbitrary (we made no assumptions constraining it), OCaml introduces a *type variable* `'a` to represent it:

```
# fun x -> x;;
- : 'a -> 'a = <fun>
```

A More Complex Example Let us try `fun x -> x+1`, which is the same as `fun x -> ((+) x) 1` (try it in OCaml!). We will use the notation $[?\alpha]$ to mean "type unknown yet, but the same as in other places marked $[?\alpha]$."

Starting the derivation and applying $\to$ introduction:

$$\frac{\frac{}{\texttt{((+) x) 1}:[?\alpha]}\,[?]}{\texttt{fun x -> ((+) x) 1} : [?] \to [?\alpha]}$$

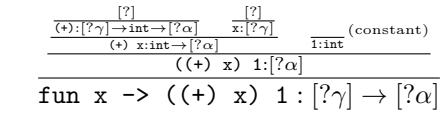
Applying $\to$ elimination (function application) to `((+) x) 1`:

$$\frac{\frac{\frac{}{\texttt{(+) x}:[?\beta]\to[?\alpha]}\,[?] \quad \frac{}{\texttt{1}:[?\beta]}\,[?]}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1} : [?] \to [?\alpha]}$$

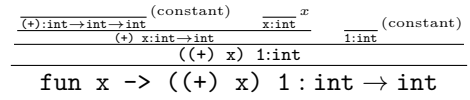
We know that `1 : int`, so $[?\beta] = \texttt{int}$:

$$\frac{\frac{\frac{}{\texttt{(+) x}:\texttt{int}\to[?\alpha]}\,[?] \quad \frac{}{\texttt{1}:\texttt{int}}\,(\text{constant})}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1} : [?] \to [?\alpha]}$$

Applying function application again to `(+) x`:



Since `(+) : int -> int -> int`, we have $[?\gamma] = \texttt{int}$ and $[?\alpha] = \texttt{int}$:



2.1.1 Curried Form When there are several arrows "on the same depth" in a function type, it means that the function returns a function. For example, `(+) : int -> int -> int` is just a shorthand for `(+) : int -> (int -> int)`. This is very different from:

$$\texttt{fun f -> (f 1) + 1} : (\texttt{int} \to \texttt{int}) \to \texttt{int}$$

In the first case, `(+)` is a function that takes an integer and returns a function from integers to integers. In the second case, we have a function that takes a function as an argument.

For addition, instead of `(fun x -> x+1)` we can write `((+) 1)`. What expanded form does `((+) 1)` correspond to exactly (computationally)? It corresponds to `fun y -> 1 + y`.

We will become more familiar with functions returning functions when we study the *lambda calculus* in a later chapter.

2.2 Algebraic Data Types

In Chapter 1, we learned about the `unit` type and variant types like:

```
type int_string_choice = A of int | B of string
```

We also covered tuple types, record types, and type definitions. Let us now explore these concepts more deeply.

Variants Without Arguments Variants do not have to carry arguments. Instead of writing `A of unit`, we can simply use `A`. This is more convenient and idiomatic:

```
type color = Red | Green | Blue
```

A subtle point about OCaml: In OCaml, variants take multiple arguments rather than taking tuples as arguments. This means `A of int * string` is different from `A of (int * string)`. The first takes two separate arguments, while the second takes a single tuple argument. This distinction is usually not important unless you encounter situations where it matters.

Recursive Type Definitions Type definitions can be recursive! This allows us to define data structures of arbitrary size:

```
type int_list = Empty | Cons of int * int_list
```

Let us see what values inhabit `int_list`: - `Empty` represents the empty list - `Cons (5, Empty)` is a list containing just 5 - `Cons (5, Cons (7, Cons (13, Empty)))` is a list containing 5, 7, and 13

The built-in type `bool` can be viewed as if it were defined as `type bool = true | false`. Similarly, `int` can be thought of as a very large variant: `type int = 0 | -1 | 1 | -2 | 2 | ...`

Parametric Type Definitions Type definitions can be *parametric* with respect to the types of their components. This allows us to define generic data structures that work with any element type. For example, a list of elements of arbitrary type:

```
type 'elem list = Empty | Cons of 'elem * 'elem list
```

Several conventions and syntax rules apply to parametric types:

- Type variables must start with `'`, but since OCaml will not remember the names we give, it is customary to use the names OCaml uses: `'a`, `'b`, `'c`, `'d`, etc.
- The OCaml syntax places the type parameter before the type name, mimicking English word order. A silly example:

  ```
  type 'white_color dog = Dog of 'white_color
  ```

- With multiple parameters, OCaml uses parentheses:

  ```
  type ('a, 'b) choice = Left of 'a | Right of 'b
  ```

  Compare this to F# syntax: `type choice<'a,'b> = Left of 'a | Right of 'b`

  And Haskell syntax: `data Choice a b = Left a | Right b`

2.3 Syntactic Conventions

Constructor Naming Names of variants, called *constructors*, must start with a capital letter. If we wanted to define our own booleans, we would write:

```
type my_bool = True | False
```

Only constructors and module names can start with capital letters in OCaml. *Modules* are organizational units (like "shelves") containing related values. For example, the `List` module provides operations on lists, including `List.map` and `List.filter`.

Accessing Record Fields We can use dot notation to access record fields: `record.field`. For example, if we have `let person = {name="Alice"; age=30}`, we can write `person.name` to get `"Alice"`.

Function Definition Shortcuts Several syntactic shortcuts make function definitions more concise:

- `fun x y -> e` stands for `fun x -> fun y -> e`. Note that `fun x -> fun y -> e` parses as `fun x -> (fun y -> e)`.
- `function A x -> e1 | B y -> e2` stands for `fun p -> match p with A x -> e1 | B y -> e2`. The general form is: `function PATTERN-MATCHING` stands for `fun v -> match v with PATTERN-MATCHING`.
- `let f ARGS = e` is a shorthand for `let f = fun ARGS -> e`.

2.4 Pattern Matching

Recall that we introduced `fst` and `snd` as means to access elements of a pair. But what about larger tuples? The fundamental way to access any tuple uses the `match` construct. In fact, `fst` and `snd` can easily be defined using pattern matching:

```
let fst = fun p -> match p with (a, b) -> a
let snd = fun p -> match p with (a, b) -> b
```

Matching on Records Pattern matching also works with records:

```
type person = {name: string; surname: string; age: int}

let greet_person () =
  match {name="Walker"; surname="Johnnie"; age=207}
  with {name=n; surname=sn; age=a} -> "Hi " ^ sn ^ "!"
```

Understanding Patterns The left-hand sides of `->` in `match` expressions are called **patterns**. Patterns describe the structure of values we want to match against.

Patterns can be nested, allowing us to match complex structures:

```
match Some (5, 7) with
| None -> "sum: nothing"
| Some (x, y) -> "sum: " ^ string_of_int (x+y)
```

Simple Patterns and Wildcards A pattern can simply bind the entire value without destructuring. Writing `match f x with v -> ...` is the same as `let v = f x in ...`.

When we do not need a value in a pattern, it is good practice to use the underscore `_`, which is a wildcard (not a variable):

```
let fst (a, _) = a
let snd (_, b) = b
```

Pattern Linearity A variable can only appear once in a pattern. This property is called *linearity*. However, we can add conditions to patterns using `when`, so linearity is not a limitation:

```
let describe_point p =
  match p with
  | (x, y) when x = y -> "diag"
  | _ -> "off-diag"
```

Here is a more elaborate example:

```
let compare a b = match a, b with
  | (x, y) when x < y -> -1
  | (x, y) when x = y -> 0
  | _ -> 1
```

Partial Record Patterns We can skip unused fields of a record in a pattern. Only the fields we care about need to be mentioned.

Or-Patterns We can compress patterns by using `|` inside a single pattern to match multiple alternatives:

```
type month =
  | Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sep | Oct | Nov | Dec

type weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

type date =
  {year: int; month: month; day: int; weekday: weekday}

let day =
  {year = 2012; month = Feb; day = 14; weekday = Wed};;

match day with
  | {weekday = Sat | Sun} -> "Weekend!"
  | _ -> "Work day"
```

Named Patterns with `as` We use `(pattern as v)` to name a nested pattern, binding the matched value to `v`:

```
match day with
```

```
| {weekday = (Mon | Tue | Wed | Thu | Fri as wday)}
    when not (day.month = Dec && day.day = 24) ->
  Some (work (get_plan wday))
| _ -> None
```

This example shows the `as` keyword binding the matched weekday to `wday` for use in the expression on the right side of the arrow.

2.5 Interpreting Algebraic Data Types as Polynomials

Let us explore a curious analogy between algebraic data types and polynomials. We translate data types to mathematical expressions by:

- Replacing `|` (variant choice) with $+$
- Replacing `*` (tuple product) with $\times$
- Treating record types as tuple types (erasing field names and translating `;` as $\times$)

We also need translations for some special types:

- The **void type** (a type with no constructors, hence no values):

  ```
  type void
  ```

  (Yes, this is its complete definition, with no `= something` part.) Translate it as 0.

- The **unit type** translates as 1. Since variants without arguments behave like variants `of unit`, translate them as 1 as well.

- The **bool type** translates as 2.

- Types like `int`, `string`, `float`, and type parameters translate as variables.

- Defined types translate according to their definitions (substituting variables as necessary).

Give a name to the type being defined (representing a function of the introduced variables). Now interpret the result as an ordinary numeric polynomial! (Or a "rational function" if recursively defined.)

Let us have some fun with this translation.

Example: Date Type

```
type date = {year: int; month: int; day: int}
```

Translating to a polynomial (using x for `int`):

$$D = x \times x \times x = x^3$$

Example: Option Type The built-in option type is defined as:

```
type 'a option = None | Some of 'a
```

Translating:

$$O = 1 + x$$

Example: List Type

```
type 'a my_list = Empty | Cons of 'a * 'a my_list
```

Translating (where L represents the list type):

$$L = 1 + x \cdot L$$

Example: Binary Tree Type

```
type btree = Tip | Node of int * btree * btree
```

Translating:

$$T = 1 + x \cdot T \cdot T = 1 + x \cdot T^2$$

Type Isomorphisms When translations of two types are equal according to the laws of high-school algebra, the types are *isomorphic*. This means there exist bijective (one-to-one and onto) functions between them.

Let us manipulate the binary tree polynomial:

$$\begin{aligned} T &= 1 + x \cdot T^2 \\ &= 1 + x \cdot T + x^2 \cdot T^3 \\ &= 1 + x + x^2 \cdot T^2 + x^2 \cdot T^3 \\ &= 1 + x + x^2 \cdot T^2 \cdot (1 + T) \\ &= 1 + x \cdot (1 + x \cdot T^2 \cdot (1 + T)) \end{aligned}$$

Now let us translate the resulting expression back to a type:

```
type repr =
  (int * (int * btree * btree * btree option) option) option
```

The challenge is to find isomorphism functions with signatures:

```
val iso1 : btree -> repr
val iso2 : repr -> btree
```

These functions should satisfy: for all trees `t`, `iso2 (iso1 t) = t`, and for all representations `r`, `iso1 (iso2 r) = r`.

A First Attempt Here is a first (failed) attempt:

```
# let iso1 (t : btree) : repr =
  match t with
    | Tip -> None
    | Node (x, Tip, Tip) -> Some (x, None)
    | Node (x, Node (y, t1, t2), Tip) ->
      Some (x, Some (y, t1, t2, None))
    | Node (x, Node (y, t1, t2), t3) ->
      Some (x, Some (y, t1, t2, Some t3));;

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Node (_, Tip, Node (_, _, _))
```

We forgot about one case! It seems difficult to guess the solution directly.

Breaking Down the Problem Let us divide the task into smaller steps corresponding to intermediate points in the polynomial transformation:

```
type ('a, 'b) choice = Left of 'a | Right of 'b

type interm1 =
  ((int * btree, int * int * btree * btree * btree) choice)
  option

type interm2 =
  ((int, int * int * btree * btree * btree option) choice)
  option
```

Now we can define each step:

```
let step1r (t : btree) : interm1 =
  match t with
    | Tip -> None
    | Node (x, t1, Tip) -> Some (Left (x, t1))
    | Node (x, t1, Node (y, t2, t3)) ->
      Some (Right (x, y, t1, t2, t3))

let step2r (r : interm1) : interm2 =
  match r with
    | None -> None
    | Some (Left (x, Tip)) -> Some (Left x)
    | Some (Left (x, Node (y, t1, t2))) ->
      Some (Right (x, y, t1, t2, None))
    | Some (Right (x, y, t1, t2, t3)) ->
      Some (Right (x, y, t1, t2, Some t3))
```

```
let step3r (r : interm2) : repr =
  match r with
    | None -> None
    | Some (Left x) -> Some (x, None)
    | Some (Right (x, y, t1, t2, t3opt)) ->
      Some (x, Some (y, t1, t2, t3opt))

let iso1 (t : btree) : repr =
  step3r (step2r (step1r t))
```

Defining `step1l`, `step2l`, `step3l`, and `iso2` is now straightforward—each step is the inverse of its corresponding forward step.

Take-Home Lessons

1. **Design for validity:** Try to define data structures so that only meaningful information can be represented—as long as it does not overcomplicate the data structures. Avoid catch-all clauses when defining functions. The compiler will then tell you if you have forgotten about a case.
2. **Divide and conquer:** Break solutions into small steps so that each step can be easily understood and verified.

2.6 Differentiating Algebraic Data Types

The title might seem strange—we will differentiate the translated polynomials, not the types themselves. But what sense does this make?

It turns out that taking the partial derivative of a polynomial (translated from a data type), when translated back, gives a type representing how to change one occurrence of a value corresponding to the variable with respect to which we differentiated. In other words, the derivative represents a "context" or "hole" in the data structure.

Example: Differentiating the Date Type

```
type date = {year: int; month: int; day: int}
```

The translation:

$$D = x \cdot x \cdot x = x^3$$
$$\frac{\partial D}{\partial x} = 3x^2 = x \cdot x + x \cdot x + x \cdot x$$

We could have left it as $3 \cdot x \cdot x$, but expanding shows the structure more clearly. Translating back to a type:

```
type date_deriv =
  Year of int * int | Month of int * int | Day of int * int
```

Each variant represents a "hole" at a different position: `Year` means the year field is missing (and we have the month and day), and so on.

Now we can define functions to introduce and eliminate this derivative type:

```
let date_deriv {year=y; month=m; day=d} =
  [Year (m, d); Month (y, d); Day (y, m)]

let date_integr n = function
  | Year (m, d) -> {year=n; month=m; day=d}
  | Month (y, d) -> {year=y; month=n; day=d}
  | Day (y, m) -> {year=y; month=m; day=n}
;;

List.map (date_integr 7)
  (date_deriv {year=2012; month=2; day=14})
```

The `date_deriv` function produces all contexts (one for each field), and `date_integr` fills in a hole with a new value.

Example: Differentiating Binary Trees Let us tackle the more challenging case of binary trees:

```
type btree = Tip | Node of int * btree * btree
```

The translation and differentiation:

$$T = 1 + x \cdot T^2$$
$$\frac{\partial T}{\partial x} = 0 + T^2 + 2 \cdot x \cdot T \cdot \frac{\partial T}{\partial x} = T \cdot T + 2 \cdot x \cdot T \cdot \frac{\partial T}{\partial x}$$

The derivative is recursive! This makes sense: a context in a tree is either at the current node ($T \cdot T$, the two subtrees) or somewhere below ($2 \cdot x \cdot T \cdot \frac{\partial T}{\partial x}$, choosing left or right, with the node value, the other subtree, and a deeper context).

Instead of translating 2 as `bool`, we introduce a more descriptive type:

```
type btree_dir = LeftBranch | RightBranch

type btree_deriv =
  | Here of btree * btree
  | Below of btree_dir * int * btree * btree_deriv
```

(You might someday hear about *zippers*—they are "inverted" relative to our type, with the hole coming first.)

The integration function fills the hole with a value:

```
let rec btree_integr n = function
  | Here (ltree, rtree) -> Node (n, ltree, rtree)
```

```
  | Below (LeftBranch, m, rtree, deriv) ->
    Node (m, btree_integr n deriv, rtree)
  | Below (RightBranch, m, ltree, deriv) ->
    Node (m, ltree, btree_integr n deriv)
```

2.7 Exercises

Exercise 1 *Due to Yaron Minsky.*

Consider a datatype to store internet connection information. The time `when_initiated` marks the start of connecting and is not needed after the connection is established (it is only used to decide whether to give up trying to connect). The ping information is available for established connections but not straight away.

```
type connectionstate = Connecting | Connected | Disconnected

type connectioninfo = {
  state : connectionstate;
  server : Inetaddr.t;
  lastpingtime : Time.t option;
  lastpingid : int option;
  sessionid : string option;
  wheninitiated : Time.t option;
  whendisconnected : Time.t option;
}
```

(The types `Time.t` and `Inetaddr.t` come from the *Core* library. You can replace them with `float` and `Unix.inet_addr`. Load the Unix library in the interactive toplevel with `#load "unix.cma";;`.)

Rewrite the type definitions so that the datatype will contain only reasonable combinations of information.

Exercise 2 In OCaml, functions can have labeled arguments and optional arguments (parameters with default values that can be omitted). Labels can differ from the names of argument values:

```
let f ~meaningfulname:n = n + 1
let _ = f ~meaningfulname:5  (* We do not need the result so we ignore it. *)
```

When the label and value names are the same, the syntax is shorter:

```
let g ~pos ~len =
  StringLabels.sub "0123456789abcdefghijklmnopqrstuvwxyz" ~pos ~len

let () =  (* A nicer way to mark computations that return unit. *)
  let pos = Random.int 26 in
```

```
  let len = Random.int 10 in
  print_string (g ~pos ~len)
```

When some function arguments are optional, the function must take non-optional arguments after the last optional argument. Optional parameters with default values:

```
let h ?(len=1) pos = g ~pos ~len
let () = print_string (h 10)
```

Optional arguments are implemented as parameters of an option type. This allows checking whether the argument was provided:

```
let foo ?bar n =
  match bar with
    | None -> "Argument = " ^ string_of_int n
    | Some m -> "Sum = " ^ string_of_int (m + n)
```

We can use it in various ways:

```
let _ = foo 5
let _ = foo ~bar:5 7
```

We can also provide the option value directly:

```
let test_foo () =
  let bar = if Random.int 10 < 5 then None else Some 7 in
  foo ?bar 7
```

1. Observe the types that functions with labeled and optional arguments have. Come up with coding style guidelines for when to use labeled arguments.
2. Write a rectangle-drawing procedure that takes three optional arguments: left-upper corner, right-lower corner, and a width-height pair. It should draw a correct rectangle whenever two arguments are given, and raise an exception otherwise. Load the graphics library with `#load "graphics.cma";;`. Use `invalid_arg`, `Graphics.open_graph`, and `Graphics.draw_rect`.
3. Write a function that takes an optional argument of arbitrary type and a function argument, and passes the optional argument to the function without inspecting it.

Exercise 3 *From a past exam.*

1. Give the (most general) types of the following expressions, either by guessing or by inferring by hand:
   1. `let double f y = f (f y) in fun g x -> double (g x)`
   2. `let rec tails l = match l with [] -> [] | x::xs -> xs::tails xs in fun l -> List.combine l (tails l)`

2. Give example expressions that have the following types (without using type constraints):
   1. `(int -> int) -> bool`
   2. `'a option -> 'a list`

Exercise 4 We have seen that algebraic data types can be related to analytic functions (the subset definable from polynomials via recursion)—by literally interpreting sum types (variant types) as sums and product types (tuple and record types) as products. We can extend this interpretation to function types by interpreting $a \to b$ as b^a (i.e., b to the power of a). Note that the b^a notation is actually used to denote functions in set theory.

1. Translate a^{b+cd} and $a^b \cdot (a^c)^d$ into OCaml types, using any distinct types for a, b, c, d, and using `type ('a,'b) choice = Left of 'a | Right of 'b` for $+$. Write the bijection function in both directions.
2. Come up with a type `'t exp` that shares with the exponential function the following property: $\frac{\partial \exp(t)}{\partial t} = \exp(t)$, where we translate a derivative of a type as a context (i.e., the type with a "hole"), as in this chapter. Explain why your answer is correct. *Hint:* in computer science, our logarithms are mostly base 2.

Further reading: Algebraic Type Systems - Combinatorial Species

Exercise 5 (Homework) Write a function `btree_deriv_at` that takes a predicate over integers (i.e., a function `f: int -> bool`) and a `btree`, and builds a `btree_deriv` whose "hole" is in the first position for which the predicate returns true. It should return a `btree_deriv option`, with `None` if the predicate does not hold for any node.

Chapter 3: Computation

Understanding how programs execute through reduction semantics

This chapter explores how OCaml programs actually compute results. We begin with function composition as a practical tool, then develop a formal model of computation using reduction semantics. The chapter concludes with important techniques for writing efficient recursive functions: tail recursion and continuation-passing style.

References: - "Using, Understanding and Unraveling the OCaml Language" by Didier Remy, chapter 1 - "The OCaml system" manual, the tutorial part, chapter 1

3.1 Function Composition

Function composition is a fundamental operation that combines two functions into a single function. In mathematics, composition is typically defined "back-

ward"—the function written first is applied last:

$$(f \circ g)(x) = f(g(x))$$

This notation follows mathematical convention where we read function application from right to left. Different functional programming languages provide operators for this style of composition:

- OCaml: `let (-|) f g x = f (g x)`
- F#: `let (<<) f g x = f (g x)`
- Haskell: `(.) f g = \x -> f (g x)`

The composition operator resembles function application but requires fewer parentheses. Recall the isomorphism functions `iso1` and `iso2` from Chapter 2, which converted between a binary tree type and an equivalent representation. Using backward composition, we can write:

```
let iso2 = step1l -| step2l -| step3l
```

Forward Composition A more intuitive definition of function composition follows the order in which computation actually proceeds—the function written first is applied first:

- OCaml: `let (|-) f g x = g (f x)`
- F#: `let (>>) f g x = g (f x)`

With forward composition, we can express the inverse isomorphism as:

```
let iso1 = step1r |- step2r |- step3r
```

Both definitions involve **partial application**—we supply only some of the arguments a function expects, obtaining a function that awaits the remaining arguments. For example, `((+) 1)` from Chapter 2 is a partial application of addition that awaits one more integer argument.

Iterated Composition We can define a function that composes a function with itself n times:

$$f^n(x) := \underbrace{(f \circ \cdots \circ f)}_{n \text{ times}}(x)$$

In OCaml, we first define the backward composition operator, then use it in `power`:

```
let (-|) f g x = f (g x)

let rec power f n =
  if n <= 0 then (fun x -> x) else f -| power f (n-1)
```

When `n` is zero or negative, we return the identity function. Otherwise, we compose `f` with the $(n-1)$-fold composition of `f`.

Numerical Differentiation As an application of function composition, let us implement numerical differentiation. The derivative of a function f at a point x can be approximated by:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

for a small value of Δx. In OCaml:

```
let derivative dx f = fun x -> (f(x +. dx) -. f(x)) /. dx
```

The explicit `fun x -> ...` emphasizes that we intend to use this with two arguments (the step size and the function to differentiate). We can write this more concisely:

```
let derivative dx f x = (f(x +. dx) -. f(x)) /. dx
```

Note that OCaml uses different operators for integer and floating-point arithmetic. The `(+)` operator has type `int -> int -> int` and cannot be used with `float` values. Instead, we use operators with a trailing dot: `+.`, `-.`, `*.`, `/.` for floating-point operations.

Now we can compute higher derivatives using our `power` function:

```
let pi = 4.0 *. atan 1.0

let sin''' = (power (derivative 1e-5) 3) sin
```

```
# sin''' pi;;
- : float = 0.999998972517346263
```

This computes the third derivative of sine (which equals $-\cos$) at π, yielding approximately 1.0.

3.2 Evaluation Rules (Reduction Semantics)

To understand precisely how OCaml programs compute, we develop a formal model called **reduction semantics**. This model specifies the step-by-step rules by which expressions are transformed until they become values.

Expressions Programs consist of **expressions**, which we define by the following grammar:

$$
\begin{array}{rll}
a ::= & x & \text{variables} \\
| & \texttt{fun}\ x\ \texttt{->}\ a & \text{(defined) functions} \\
| & a\ a & \text{applications} \\
| & C^0 & \text{value constructors of arity 0} \\
| & C^n(a, \ldots, a) & \text{value constructors of arity } n \\
| & f^n & \text{built-in values (primitives) of arity } n \\
| & \texttt{let}\ x = a\ \texttt{in}\ a & \text{name bindings (local definitions)} \\
| & \texttt{match}\ a\ \texttt{with} & \\
& \quad p\ \texttt{->}\ a\ \texttt{|}\ \ldots\ \texttt{|}\ p\ \texttt{->}\ a & \text{pattern matching}
\end{array}
$$

Patterns are defined as:

$$
\begin{array}{rll}
p ::= & x & \text{pattern variables} \\
| & (p, \ldots, p) & \text{tuple patterns} \\
| & C^0 & \text{variant patterns of arity 0} \\
| & C^n(p, \ldots, p) & \text{variant patterns of arity } n
\end{array}
$$

The **arity** of a constructor or primitive indicates how many arguments it requires. For tuples, arity corresponds to the tuple's length.

Recursion via fix To simplify our presentation, we introduce a primitive `fix` that captures the essence of recursive definitions. A limited form of `let rec` can be expressed using `fix`:

$$
\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 \equiv \texttt{let}\ f = \texttt{fix (fun}\ f\ x\ \texttt{->}\ e_1\texttt{)}\ \texttt{in}\ e_2
$$

The `fix` primitive is a **fixed-point combinator**—it finds a fixed point of a function, allowing recursion without explicit self-reference in the syntax.

Values Expressions evaluate (compute) to **values**:

$$
\begin{array}{rll}
v ::= & \texttt{fun}\ x\ \texttt{->}\ a & \text{(defined) functions} \\
| & C^n(v_1, \ldots, v_n) & \text{constructed values} \\
| & f^n\ v_1\ \ldots\ v_k & k < n \text{ (partially applied primitives)}
\end{array}
$$

Values are expressions that cannot be reduced further. Functions are values (we do not evaluate under the lambda). Constructed values have all their components fully evaluated. Partially applied primitives are values because they still await more arguments.

Substitution To describe computation, we need the notion of **substitution**. Writing $a[x := v]$ means substituting value v for variable x in expression a—every occurrence of x in a is replaced by v.

In practice, implementations do not actually copy v for each occurrence; they use references or environments. But the substitution model gives the correct semantics.

Reduction Rules (Redexes) Reduction proceeds by transforming **redexes** (reducible expressions) according to these rules:

$$\begin{aligned}
(\texttt{fun}\ x\ \texttt{->}\ a)\ v &\rightsquigarrow a[x := v] && \text{(function application)}\\
\texttt{let}\ x = v\ \texttt{in}\ a &\rightsquigarrow a[x := v] && \text{(let binding)}\\
f^n\ v_1\ \ldots\ v_n &\rightsquigarrow f(v_1, \ldots, v_n) && \text{(primitive application)}
\end{aligned}$$

For pattern matching:

$$\texttt{match}\ v\ \texttt{with}\ x\ \texttt{->}\ a\ \texttt{|}\ \ldots \rightsquigarrow a[x := v]$$

When matching a constructor pattern:

$$\begin{aligned}
&\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}\ C_2^k(p_1, \ldots, p_k)\ \texttt{->}\ a\ \texttt{|}\ pm\\
&\quad\rightsquigarrow \texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}\ pm
\end{aligned}$$

(when $C_1 \neq C_2$ or the arities differ, we try the next pattern)

$$\begin{aligned}
&\texttt{match}\ C_1^n(v_1, \ldots, v_n)\ \texttt{with}\ C_1^n(x_1, \ldots, x_n)\ \texttt{->}\ a\ \texttt{|}\ \ldots\\
&\quad\rightsquigarrow a[x_1 := v_1; \ldots; x_n := v_n]
\end{aligned}$$

When $n = 0$, the notation $C_1^n(v_1, \ldots, v_n)$ simply means C_1^0 (a nullary constructor). By $f(v_1, \ldots, v_n)$, we denote the actual value resulting from computing the primitive. We omit more complex cases of pattern matching (such as nested patterns) for simplicity.

Note on rule variables: In these rules, x matches any expression or pattern variable; $a, a_1, \ldots, a_n$ match any expression; $v, v_1, \ldots, v_n$ match any value. To apply a rule, substitute the rule variables so that the left-hand side matches your expression, then the right-hand side is the reduced expression.

Context Rules The reduction rules above only apply when arguments are already values. The remaining rules evaluate subexpressions. If $a_i \rightsquigarrow a_i'$, then:

$$a_1\ a_2 \rightsquigarrow a_1'\ a_2$$
$$a_1\ a_2 \rightsquigarrow a_1\ a_2'$$
$$C^n(a_1, \ldots, a_i, \ldots, a_n) \rightsquigarrow C^n(a_1, \ldots, a_i', \ldots, a_n)$$
$$\texttt{let}\ x = a_1\ \texttt{in}\ a_2 \rightsquigarrow \texttt{let}\ x = a_1'\ \texttt{in}\ a_2$$
$$\texttt{match}\ a_1\ \texttt{with}\ pm \rightsquigarrow \texttt{match}\ a_1'\ \texttt{with}\ pm$$

Arguments can be evaluated in arbitrary order, but `let...in` and `match...with` maintain their sequential structure: we evaluate the bound expression or scrutinee before the body or branches.

The fix Primitive Finally, we give the reduction rule for `fix`, which is a binary primitive:

$$\texttt{fix}^2\ v_1\ v_2 \rightsquigarrow v_1\ (\texttt{fix}^2\ v_1)\ v_2$$

Because `fix` is binary, $(\texttt{fix}^2\ v_1)$ is already a value (a partially applied primitive). It will not be further computed until it is applied inside v_1. This delayed evaluation is what makes recursion work—the recursive call is only expanded when actually needed.

Exercise: Compute some simple programs by hand using these reduction rules. For example, trace through `(fun x -> x + 1) 5` or `let double f x = f (f x) in double (fun n -> n * 2) 3`.

3.3 Symbolic Derivation Example

To see reduction in action, consider the symbolic expression evaluator from the `Lec3.ml` file. We define an algebraic data type for mathematical expressions:

```
type expression =
   Const of float
 | Var of string
 | Sum of expression * expression    (* e1 + e2 *)
 | Diff of expression * expression   (* e1 - e2 *)
 | Prod of expression * expression   (* e1 * e2 *)
 | Quot of expression * expression   (* e1 / e2 *)
```

We can evaluate expressions given an environment mapping variable names to values:

```
exception Unbound_variable of string

let rec eval env exp =
   match exp with
     Const c -> c
   | Var v ->
```

```
      (try List.assoc v env with Not_found -> raise(Unbound_variable v))
  | Sum(f, g) -> eval env f +. eval env g
  | Diff(f, g) -> eval env f -. eval env g
  | Prod(f, g) -> eval env f *. eval env g
  | Quot(f, g) -> eval env f /. eval env g
```

And compute symbolic derivatives:

```
let rec deriv exp dv =
  match exp with
    Const c -> Const 0.0
  | Var v -> if v = dv then Const 1.0 else Const 0.0
  | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
  | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
  | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
  | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
                       Prod(g, g))
```

For convenience, we define helper operators:

```
let x = Var "x"
let y = Var "y"
let (+:) f g = Sum (f, g)
let (-:) f g = Diff (f, g)
let ( *: ) f g = Prod (f, g)
let (/:) f g = Quot (f, g)
let (!:) i = Const i

let example = !:3.0 *: x +: !:2.0 *: y +: x *: x *: y
```

When we trace the evaluation of `example` with $x = 1$ and $y = 2$, we can observe the recursive structure of computation. The trace shows how subexpressions are evaluated depth-first:

```
eval_1_2 <-- 3.00 * x + 2.00 * y + x * x * y
  eval_1_2 <-- x * x * y
    eval_1_2 <-- y
    eval_1_2 --> 2.
    eval_1_2 <-- x * x
      eval_1_2 <-- x
      eval_1_2 --> 1.
      eval_1_2 <-- x
      eval_1_2 --> 1.
    eval_1_2 --> 1.
  eval_1_2 --> 2.
  eval_1_2 <-- 3.00 * x + 2.00 * y
    eval_1_2 <-- 2.00 * y
      eval_1_2 <-- y
      eval_1_2 --> 2.
```

```
      eval_1_2 <-- 2.00
      eval_1_2 --> 2.
    eval_1_2 --> 4.
    eval_1_2 <-- 3.00 * x
      eval_1_2 <-- x
      eval_1_2 --> 1.
      eval_1_2 <-- 3.00
      eval_1_2 --> 3.
    eval_1_2 --> 3.
  eval_1_2 --> 7.
eval_1_2 --> 9.
- : float = 9.
```

The indentation levels correspond roughly to stack frames in the computer's execution. Each `<--` marks entering a recursive call; each `-->` marks returning with a result. This trace visualization helps us understand the connection between formal reduction rules and actual program execution.

3.4 Tail Calls and Tail Recursion

Understanding how function calls work in practice is essential for writing efficient recursive programs.

When a function calls another function (or itself), the computer typically creates a **stack frame** to store information about the call: the return address, local variables, and so on. The indentation levels in the evaluation trace above roughly correspond to these stack frames.

A **tail call** is a function call that is the last action performed by a function—there is no more work to do after the call returns. When a tail call occurs, the current stack frame is no longer needed; we can reuse its space for the callee's frame.

Functional language compilers optimize tail calls by inserting a "jump" instruction instead of creating a new stack frame. This optimization is called **tail call elimination** or **tail call optimization**.

A function is **tail recursive** if it calls itself (and any mutually recursive functions) only in tail position. Tail recursive functions can run in constant stack space, regardless of how many iterations they perform.

Non-Tail-Recursive Example Consider this function that generates a list counting down from `n`:

```
let rec unfold n = if n <= 0 then [] else n :: unfold (n-1)
```

The recursive call `unfold (n-1)` is not in tail position because we still need to cons `n` onto the result after the call returns. Each recursive call creates a new stack frame. In a system with limited stack space:

```
# unfold 100000;;
```

```
- : int list =
[100000; 99999; 99998; 99997; 99996; 99995; 99994; 99993; ...]

# unfold 1000000;;
Stack overflow during evaluation (looping recursion?).
```

With 100,000 iterations, the function succeeds. With 1,000,000, we may run out of stack space (depending on system configuration).

Tail-Recursive Version with Accumulator We can rewrite the function to be tail recursive by using an **accumulator** argument that stores intermediate results:

```
let rec unfold_tcall acc n =
  if n <= 0 then acc else unfold_tcall (n::acc) (n-1)
```

Now the recursive call is the last thing the function does—there is no work remaining after it returns. The result is built up in the accumulator as we recurse:

```
# unfold_tcall [] 100000;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; ...]

# unfold_tcall [] 1000000;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; ...]
```

The tail-recursive version handles 1,000,000 iterations without stack overflow. Notice that the result is in ascending order rather than descending—the accumulated result is computed in "reverse order," building up while climbing the recursion rather than while returning from it.

Tail Recursion and Data Structures Can every recursive function be made tail recursive? Consider computing the depth of a binary tree (using the `btree` type from Chapter 2):

```
type btree = Tip | Node of int * btree * btree
```

The naive approach to computing depth:

```
let rec depth_naive tree = match tree with
  | Tip -> 0
  | Node (_, left, right) -> 1 + max (depth_naive left) (depth_naive right)
```

We need to compute both `depth_naive left` and `depth_naive right` before we can compute their maximum. This seems inherently non-tail-recursive—or is it?

Note: The situation is more complex for lazy programming languages like Haskell, where evaluation is demand-driven and different optimization strategies apply.

3.5 First Encounter of Continuation-Passing Style

Continuation-passing style (CPS) is a programming technique that makes control flow explicit. Instead of returning a value directly, a CPS function takes an extra argument—a **continuation**—that represents "what to do next" with the result.

Using CPS, we can transform any recursive function into a tail-recursive one. The key insight is that we postpone the actual work until the very end, building up a chain of continuations that encode the remaining computation.

Here is the tree depth function in CPS:

```
let rec depth_cps tree k = match tree with
    | Tip -> k 0
    | Node(_, left, right) ->
      depth_cps left (fun dleft ->
        depth_cps right (fun dright ->
          k (1 + (max dleft dright))))

let depth tree = depth_cps tree (fun d -> d)
```

Let us trace through how this works:

1. For `Tip`, we immediately call the continuation with 0.
2. For `Node`, we recursively compute the depth of the left subtree, passing a continuation that will:
   - Recursively compute the depth of the right subtree, passing a continuation that will:
     - Combine the results and pass to the original continuation.

The outer `depth` function calls the CPS version with the identity continuation `(fun d -> d)`, which simply returns the final result.

We can test it:

```
# let example_tree =
    Node(1, Node(2, Node(3, Tip, Tip), Tip), Node(4, Tip, Tip));;
val example_tree : btree =
  Node (1, Node (2, Node (3, Tip, Tip), Tip), Node (4, Tip, Tip))
# depth example_tree;;
- : int = 3
```

Every recursive call is now in tail position! The "stack" of pending computations is represented explicitly as nested closures (the continuation functions). These

closures are allocated on the heap rather than the stack, so we avoid stack overflow.

CPS is a powerful technique with applications beyond tail recursion:

- **Implementing control operators** like exceptions, coroutines, and backtracking
- **Compiler intermediate representations** that make control flow explicit
- **Asynchronous programming** where continuations represent callbacks

We will explore CPS more deeply in later chapters.

3.6 Exercises

By "traverse a tree" below we mean: write a function that takes a tree and returns a list of values in the nodes of the tree.

Exercise 1: Write a function of type `btree -> int list` that traverses a binary tree:

1. In prefix order—first the value stored in a node, then values in all nodes to the left, then values in all nodes to the right.
2. In infix order—first values in all nodes to the left, then value stored in a node, then values in all nodes to the right (so it is "left-to-right" order).
3. In breadth-first order—first values in more shallow nodes.

Exercise 2: Turn the function from Exercise 1 (point 1 or 2) into continuation-passing style.

Exercise 3: Do the homework from the end of Chapter 2: write `btree_deriv_at` that takes a predicate over integers (i.e., a function `f: int -> bool`) and a `btree`, and builds a `btree_deriv` whose "hole" is in the first position for which the predicate returns true. It should return a `btree_deriv option`, with `None` if the predicate does not hold for any node.

Exercise 4: Write a function `simplify: expression -> expression` that simplifies the expression a bit, so that for example the result of `simplify (deriv exp dv)` looks more like what a human would get computing the derivative of `exp` with respect to `dv`.

Hint: Write a `simplify_once` function that performs a single step of the simplification, and wrap it using a general `fixpoint` function that performs an operation until a *fixed point* is reached: given f and x, it computes $f^n(x)$ such that $f^n(x) = f^{n+1}(x)$.

Exercise 5: Write two sorting algorithms working on lists: merge sort and quicksort.

1. Merge sort splits the list roughly in half, sorts the parts, and merges the sorted parts into the sorted result.

2. Quicksort splits the list into elements smaller/greater than the first element (the pivot), sorts the parts, and puts them together.