The Expression Problem

Code organization, extensibility and reuse

by Łukasz Stafiniak

Email: `lukstafi@gmail.com`
Web: `www.ii.uni.wroc.pl/~lukstafi`

- Ralf Lämmel lectures on MSDN's Channel 9:
  The Expression Problem, Haskell's Type Classes
- The old book *Developing Applications with Objective Caml*:
  Comparison of Modules and Objects, Extending Components
- The new book *Real World OCaml*: Chapter 11: Objects, Chapter 12: Classes
- Jacques Garrigue's Code reuse through polymorphic variants,
  and Recursive Modules for Programming with Keiko Nakata
- Extensible variant types
- Graham Hutton's and Erik Meijer's Monadic Parser Combinators

The Expression Problem: Definition

- The *Expression Problem*: design an implementation for expressions, where:
  - new variants of expressions can be added (*datatype extensibility*),
  - new operations on the expressions can be added (*functional extensibility*).
- By *extensibility* we mean three conditions:
  - code-level modularization: the new datatype variants, and new operations, are in separate files,
  - separate compilation: the files can be compiled and distributed separately,
  - static type safety: we do not lose the type checking help and guarantees.
- The name comes from an example: extend a language of expressions with new constructs:
  - lambda calculus: variables `Var`, λ-abstractions `Abs`, function applications `App`;
  - arithmetics: variables `Var`, constants `Num`, addition `Add`, multiplication `Mult`; …

  and new operations:
  - evaluation `eval`;
  - pretty-printing to strings `string_of`;
  - free variables `free_vars`; …

Functional Programming Non-solution: ordinary Algebraic Datatypes

- Pattern matching makes functional extensibility easy in functional programming.
- Ensuring datatype extensibility is complicated when using standard variant types.
- For brevity, we will place examples in a single file, but the component type and function definitions are not mutually recursive so can be put in separate modules.
- Non-solution penalty points:
  - Functions implemented for a broader language (e.g. `lexpr_t`) cannot be used with a value from a narrower langugage (e.g. `expr_t`).
  - Significant memory (and some time) overhead due to so called *tagging*: work of the `wrap` and `unwrap` functions, adding tags e.g. `Lambda` and `Expr`.
  - Some code bloat due to tagging. For example, deep pattern matching needs to be manually unrolled and interspersed with calls to `unwrap`.

  Verdict: non-solution, but better than extensible variant types-based approach (next) and direct OOP approach (later).

```
type var = string
```

Variables constitute a sub-language of its own.

We treat this sub-language slightly differently – no need for a dedicated variant.

```
let eval_var wrap sub (s : var) =
  try List.assoc s sub with Not_found -> wrap s
```

```
type 'a lambda =
  VarL of var |Abs of string * 'a |App of 'a * 'a

let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n

let eval_lambda eval_rec wrap unwrap subst e =
  match unwrap e with
  | Some (VarL v) -> eval_var (fun v -> wrap (VarL v)) subst v
  | Some (App (l1, l2)) ->
    let l1' = eval_rec subst l1
    and l2' = eval_rec subst l2 in
    (match unwrap l1' with
    | Some (Abs (s, body)) ->
      eval_rec [s, l2'] body
    | _ -> wrap (App (l1', l2')))
  | Some (Abs (s, l1)) ->
    let s' = gensym () in
    wrap (Abs (s', eval_rec ((s, wrap (VarL s'))::subst) l1))
  | None -> e

type lambda_t = Lambda_t of lambda_t lambda

let rec eval1 subst =
```

Here we define the sub-language of λ-expressions.

During evaluation, we need to freshen variables to avoid capture (mistaking distinct variables with the same name).

Alternatively, unwrapping could use an exception, but we use the option type as it is safer and more flexible in this context.

Recursive processing function returns expression of the completed language, we need to unwrap it into the current sub-language.

The recursive call is already wrapped.

Wrap into the completed language.

Rename variable to avoid capture (α-equivalence).

Falling-through when not in the current sub-language.

Defining λ-expressions as the completed language, and the corresponding `eval` function.

```
  eval_lambda eval1
    (fun e -> Lambda_t e) (fun (Lambda_t e) -> Some e) subst
```

The sub-language of arithmetic expressions.

```
type 'a expr =
  VarE of var | Num of int | Add of 'a * 'a | Mult of 'a * 'a

let eval_expr eval_rec wrap unwrap subst e =
  match unwrap e with
  | Some (Num _) -> e
  | Some (VarE v) ->
    eval_var (fun x -> wrap (VarE x)) subst v
  | Some (Add (m, n)) ->
    let m' = eval_rec subst m
    and n' = eval_rec subst n in
    (match unwrap m', unwrap n' with
    | Some (Num m'), Some (Num n') ->
      wrap (Num (m' + n'))
    | _ -> wrap (Add (m', n')))
```

Unwrapping to check if the subexpressions got computed to values.

Here `m'` and `n'` are wrapped.

```
  | Some (Mult (m, n)) ->
    let m' = eval_rec subst m
    and n' = eval_rec subst n in
    (match unwrap m', unwrap n' with
    | Some (Num m'), Some (Num n') ->
```

```
      wrap (Num (m' * n'))
    | _ -> wrap (Mult (m', n')))
  | None -> e
```

```
type expr_t = Expr_t of expr_t expr
```

Defining arithmetic expressions as the completed language, aka. "tying the recursive knot".

```
let rec eval2 subst =
  eval_expr eval2
    (fun e -> Expr_t e) (fun (Expr_t e) -> Some e) subst
```

```
type 'a lexpr =
  Lambda of 'a lambda | Expr of 'a expr
```

The language merging λ-expressions and arithmetic expressions, can also be used as a sub-language for further extensions.

```
let eval_lexpr eval_rec wrap unwrap subst e =
  eval_lambda eval_rec
    (fun e -> wrap (Lambda e))
    (fun e ->
      match unwrap e with
      | Some (Lambda e) -> Some e
      | _ -> None)
    subst
    (eval_expr eval_rec
       (fun e -> wrap (Expr e))
```

We use the "fall-through" property of `eval_expr` to combine the evaluators.

```
      (fun e ->
        match unwrap e with
        | Some (Expr e) -> Some e
        | _ -> None)
      subst e)

type lexpr_t = LExpr_t of lexpr_t lexpr
let rec eval3 subst =
  eval_lexpr eval3
    (fun e -> LExpr_t e)
    (fun (LExpr_t e) -> Some e) subst
```

Tying the recursive knot one last time.

- Exceptions have always formed an extensible variant type in OCaml, whose pattern matching is done using the `try...with` syntax. Since recently, new extensible variant types can be defined. This augments the normal function extensibility of FP with straightforward data extensibility.
- Non-solution penalty points:
  - Giving up exhaustivity checking, which is an important aspect of static type safety.
  - More natural with "single inheritance" extension chains, although merging is possible, and demonstrated in our example.
  - Requires "tying the recursive knot" for functions.

  Verdict: pleasant-looking, but the worst approach because of possible bugginess. Unless bug-proneness is not a concern, then the best approach.

```
type expr = ..
type var_name = string
type expr += Var of string
```

This is how extensible variant types are defined.

We add a variant case.

```
let eval_var sub = function
  | Var s as v -> (try List.assoc s sub with Not_found -> v)
  | e -> e
```

```ocaml
let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n

type expr += Abs of string * expr | App of expr * expr
```

The sub-languages are not differentiated by types, a shortcoming of this non-solution.

```ocaml
let eval_lambda eval_rec subst = function
  | Var _ as v -> eval_var subst v
  | App (l1, l2) ->
    let l2' = eval_rec subst l2 in
    (match eval_rec subst l1 with
    | Abs (s, body) ->
      eval_rec [s, l2'] body
    | l1' -> App (l1', l2'))
  | Abs (s, l1) ->
    let s' = gensym () in
    Abs (s', eval_rec ((s, Var s')::subst) l1)
  | e -> e

let freevars_lambda freevars_rec = function
  | Var v -> [v]
  | App (l1, l2) -> freevars_rec l1 @ freevars_rec l2
  | Abs (s, l1) ->
    List.filter (fun v -> v <> s) (freevars_rec l1)
  | _ -> []
```

```
let rec eval1 subst e = eval_lambda eval1 subst e
let rec freevars1 e = freevars_lambda freevars1 e
let test1 = App (Abs ("x", Var "x"), Var "y")
let e_test = eval1 [] test1
let fv_test = freevars1 test1

type expr += Num of int | Add of expr * expr | Mult of expr * expr

let map_expr f = function
  | Add (e1, e2) -> Add (f e1, f e2)
  | Mult (e1, e2) -> Mult (f e1, f e2)
  | e -> e

let eval_expr eval_rec subst e =
  match map_expr (eval_rec subst) e with
  | Add (Num m, Num n) -> Num (m + n)
  | Mult (Num m, Num n) -> Num (m * n)
  | (Num _ | Add _ | Mult _) as e -> e
  | e -> e

let freevars_expr freevars_rec = function
  | Num _ -> []
```

```
  | Add (e1, e2) | Mult (e1, e2) -> freevars_rec e1 @ freevars_rec e2
  | _ -> []

let rec eval2 subst e = eval_expr eval2 subst e
let rec freevars2 e = freevars_expr freevars2 e
let test2 = Add (Mult (Num 3, Var "x"), Num 1)
let e_test2 = eval2 [] test2
let fv_test2 = freevars2 test2

let eval_lexpr eval_rec subst e =
  eval_expr eval_rec subst (eval_lambda eval_rec subst e)
let freevars_lexpr freevars_rec e =
  freevars_lambda freevars_rec e @ freevars_expr freevars_rec e

let rec eval3 subst e = eval_lexpr eval3 subst e
let rec freevars3 e = freevars_lexpr freevars3 e
let test3 =
  App (Abs ("x", Add (Mult (Num 3, Var "x"), Num 1)),
       Num 2)
let e_test3 = eval3 [] test3
let fv_test3 = freevars3 test3
```

- OCaml's *objects* are values, somewhat similar to records.
- Viewed from the outside, an OCaml object has only *methods*, identifying the code with which to respond to messages, i.e. method invocations.
- All methods are *late-bound*, the object determines what code is run (i.e. *virtual* in C++ parlance).
- *Subtyping* determines if an object can be used in some context. OCaml has *structural subtyping*: the content of the types concerned decides if an object can be used.
- Parametric polymorphism can be used to infer if an object has the required methods.

```
let f x = x#m
val f : < m : 'a; .. > -> 'a
```

Method invocation: object#method.
Type poymorphic in two ways: 'a is the method type,
.. means that objects with more methods will be accepted.

- Methods are computed when they are invoked, even if they do not take arguments.
- We define objects inside `object...end` (compare: records `{...}`) using keywords `method` for methods, `val` for constant fields and `val mutable` for mutable fields. Constructor arguments can often be used instead of constant fields:

```
let square w = object
  method area = float_of_int (w * w) method width = w end
```

- Subtyping often needs to be explicit: we write (object :> supertype) or in more complex cases (object : type :> supertype).
  - Technically speaking, subtyping in OCaml always is explicit, and *open types*, containing .., use *row polymorphism* rather than subtyping.

```
let a = object method m = 7  method x = "a" end
let b = object method m = 42 method y = "b" end
let l = [a; b]
Error: This expression has type < m : int; y : string >
       but an expression was expected of type < m : int; x : string >
       The second object type has no method y
let l = [(a :> <m : 'a>); (b :> <m : 'a>)]
val l : < m : int > list
```

Toy example: object types share some but not all methods.
The exact types of the objects do not agree.
But the types share a supertype.

- *Variance* determines how type parameters behave wrt. subtyping:
  - *Invariant parameters* cannot be subtyped:

```
let f x = (x : <m : int; n : float> array :> <m : int> array)
Error: Type < m : int; n : float > array is not a subtype of
         < m : int > array
       The second object type has no method n
```

  - *Covariant parameters* are subtyped in the same direction as the type:

```
let f x = (x : <m : int; n : float> list :> <m : int> list)
val f : < m : int; n : float > list -> < m : int > list
```

- *Contravariant parameters* are subtyped in the opposite direction:

```
let f x = (x : <m : int; n : float> -> float :> <m : int> -> float)
Error: Type < m : int; n : float > -> float is not a subtype of
         < m : int > -> float
       Type < m : int > is not a subtype of < m : int; n : float >
let f x = (x : <m : int> -> float :> <m : int; n : float> -> float)
val f : (< m : int > -> float) -> < m : int; n : float > -> float
```

- The system of object classes in OCaml is similar to the module system.
  - Object classes are not types. Classes are a way to build object *constructors* – functions that return objects.
  - Classes have their types (compare: modules and signatures).
- In OCaml parlance:
  - late binding is not called anything – all methods are late-bound (in C++ called virtual)
  - a method or field declared to be defined in sub-classes is *virtual* (in C++ called abstract); classes that use virtual methods or fields are also called virtual
  - a method that is only visible in sub-classes is *private* (in C++ called protected)
  - a method not visible outside the class is not called anything (in C++ called private) – provide the type for the class, and omit the method in the class type (compare: module signatures and `.mli` files)
- OCaml allows multiple inheritance, which can be used to implement *mixins* as virtual / abstract classes.
- Inheritance works somewhat similarly to textual inclusion.
- See the excellent examples in https://realworldocaml.org/v1/en/html/classes.html
- You can perform `ocamlc -i Objects.ml` etc. to see inferred object and class types.

- It turns out that although object oriented programming was designed with data extensibility in mind, it is a bad fit for recursive types, like in the expression problem. Below is my attempt at solving our problem using classes – can you do better?
- Non-solution penalty points:
  - Functions implemented for a broader language (e.g. corresponding to `lexpr_t` on other slides) cannot handle values from a narrower one (e.g. corresponding to `expr_t`).
  - Writing a new function requires extending the language.
  - No deep pattern matching.

  Verdict: non-solution, better only than the extensible variant types-based approach.

```
type var_name = string
let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n

class virtual ['lang] evaluable =
object
  method virtual eval : (var_name * 'lang) list -> 'lang
  method virtual rename : var_name -> var_name -> 'lang
  method apply (_arg : 'lang)
    (fallback : unit -> 'lang) (_subst : (var_name * 'lang) list) =
    fallback ()
end

class ['lang] var (v : var_name) =
```

The abstract class for objects supporting the `eval` method. For λ-calculus, we need helper functions: renaming of free variables, β-reduction if possible (fallback otherwise).

```
object (self)                                                    We name the current object self.
  inherit ['lang] evaluable
  val v = v
  method eval subst =
    try List.assoc v subst with Not_found -> self
  method rename v1 v2 =                                                     Renaming a variable:
    if v = v1 then {< v = v2 >} else self       we clone the current object putting the new name.
end

class ['lang] abs (v : var_name) (body : 'lang) =
object (self)
  inherit ['lang] evaluable
  val v = v
  val body = body
  method eval subst =                                   We do α-conversion prior to evaluation.
    let v' = gensym () in                                    Alternatively, we could evaluate with
    {< v = v'; body = (body#rename v v')#eval subst >}                            substitution of v
  method rename v1 v2 =                           by v_inst v' : 'lang similar to num_inst below.
    if v = v1 then self                           Renaming the free variable v1, so no work if v=v1.
    else {< body = body#rename v1 v2 >}
  method apply arg _ subst =
    body#eval ((v, arg)::subst)
end

class ['lang] app (f : 'lang) (arg : 'lang) =
object (self)
  inherit ['lang] evaluable
  val f = f
```

```
  val arg = arg
  method eval subst =
    let arg' = arg#eval subst in
    f#apply arg' (fun () -> {< f = f#eval subst; arg = arg' >}) subst
  method rename v1 v2 =
    {< f = f#rename v1 v2; arg = arg#rename v1 v2 >}
end
```

We use `apply` to differentiate between `f` = `abs` (β-redexes) and `f` $\neq$ `abs`.

Cloning the object ensures that it will be a subtype of `'lang` rather than just `'lang app`.

```
type evaluable_t = evaluable_t evaluable
let new_var1 v : evaluable_t = new var v
let new_abs1 v (body : evaluable_t) : evaluable_t = new abs v body
```

These definitions only add nice-looking types.

```
class virtual compute_mixin = object
  method compute : int option = None
end
```

For evaluating arithmetic expressions we need a heper method `compute`.

```
class ['lang] var_c v = object
  inherit ['lang] var v
  inherit compute_mixin
end
class ['lang] abs_c v body = object
  inherit ['lang] abs v body
  inherit compute_mixin
end
class ['lang] app_c f arg = object
  inherit ['lang] app f arg
  inherit compute_mixin
end
```

To use λ-expressions together with arithmetic expressions we need to upgrade them with the helper method.

A numerical constant.

```
class ['lang] num (i : int) =
object (self)
  inherit ['lang] evaluable
  val i = i
  method eval _subst = self
  method rename _ _ = self
  method compute = Some i
end
```

Abstract class for evaluating arithmetic operations.

```
class virtual ['lang] operation
    (num_inst : int -> 'lang) (n1 : 'lang) (n2 : 'lang) =
object (self)
  inherit ['lang] evaluable
  val n1 = n1
  val n2 = n2
  method eval subst =
    let self' = {< n1 = n1#eval subst; n2 = n2#eval subst >} in
    match self'#compute with
    | Some i -> num_inst i
    | _ -> self'
  method rename v1 v2 = {< n1 = n1#rename v1 v2; n2 = n2#rename v1 v2 >}
end
```

We need to inject the integer as a constant that is a subtype of `'lang`.

```
class ['lang] add num_inst n1 n2 =
object (self)
  inherit ['lang] operation num_inst n1 n2
  method compute =
    match n1#compute, n2#compute with
```

If compute is called by `eval`, as intended, then `n1` and `n2` are already computed.

```
    | Some i1, Some i2 -> Some (i1 + i2)
    | _ -> None
end

class ['lang] mult num_inst n1 n2 =
object (self)
  inherit ['lang] operation num_inst n1 n2
  method compute =
    match n1#compute, n2#compute with
    | Some i1, Some i2 -> Some (i1 * i2)
    | _ -> None
end

class virtual ['lang] computable =
object
  inherit ['lang] evaluable
  inherit compute_mixin
end
```

This class is defined merely to provide an object type, we could also define this object type "by hand".

```
type computable_t = computable_t computable
let new_var2 v : computable_t = new var_c v
let new_abs2 v (body : computable_t) : computable_t = new abs_c v body
let new_app2 v (body : computable_t) : computable_t = new app_c v body
let new_num2 i : computable_t = new num i
let new_add2 (n1 : computable_t) (n2 : computable_t) : computable_t =
  new add new_num2 n1 n2
let new_mult2 (n1 : computable_t) (n2 : computable_t) : computable_t =
  new mult new_num2 n1 n2
```

Nice types for all the constructors.

OOP: The Visitor Pattern

- The *Visitor Pattern* is an object-oriented programming pattern for turning objects into variants with shallow pattern-matching (i.e. dispatch based on which variant a value is). It replaces data extensibility by operation extensibility.
- I needed to use imperative features (mutable fields), can you do better?
- Penalty points:
  - Heavy code bloat.
  - Side-effects appear to be required.
  - No deep pattern matching.

  Verdict: poor solution, better than approaches we considered so far, and worse than approaches we consider next.

```
type 'visitor visitable = < accept : 'visitor -> unit >
```

The variants need be visitable.

We store the computation as side effect because of the difficulty to keep the visitor polymorphic but have the result type depend on the visitor.

```
type var_name = string
```

```
class ['visitor] var (v : var_name) =
object (self)
  method v = v
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitVar self
end
```

The `'visitor` will determine the (sub)language to which a given `var` variant belongs.

The visitor pattern inverts the way pattern matching proceeds: the variant selects the pattern matching branch.

```
let new_var v = (new var v :> 'a visitable)
```

Visitors need to see the stored data, but distinct constructors need to belong to the same type.

```
class ['visitor] abs (v : var_name) (body : 'visitor visitable) =
object (self)
  method v = v
  method body = body
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitAbs self
end
let new_abs v body = (new abs v body :> 'a visitable)

class ['visitor] app (f : 'visitor visitable) (arg : 'visitor visitable) =
object (self)
  method f = f
  method arg = arg
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitApp self
end
let new_app f arg = (new app f arg :> 'a visitable)

class virtual ['visitor] lambda_visit =
object
  method virtual visitVar : 'visitor var -> unit
  method virtual visitAbs : 'visitor abs -> unit
  method virtual visitApp : 'visitor app -> unit
end
```

This abstract class has two uses: it defines the visitors for the sub-langauge of λ-expressions, and it will provide an early check that the visitor classes implement all the methods.

```
let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n
```

```
class ['visitor] eval_lambda
  (subst : (var_name * 'visitor visitable) list)
  (result : 'visitor visitable ref) =
object (self)
  inherit ['visitor] lambda_visit
  val mutable subst = subst
  val mutable beta_redex : (var_name * 'visitor visitable) option = None
  method visitVar var =
    beta_redex <- None;
    try result := List.assoc var#v subst
    with Not_found -> result := (var :> 'visitor visitable)
  method visitAbs abs =
    let v' = gensym () in
    let orig_subst = subst in
    subst <- (abs#v, new_var v')::subst;
    (abs#body)#accept self;
    let body' = !result in
    subst <- orig_subst;
    beta_redex <- Some (v', body');
    result := new_abs v' body'
  method visitApp app =
    app#arg#accept self;
    let arg' = !result in
    app#f#accept self;
    let f' = !result in
    match beta_redex with
    | Some (v', body') ->
```

An output argument, but also used internally to store intermediate results.

We avoid threading the argument through the visit methods.

We work around the need to differentiate between abs and non-abs values of app#f inside visitApp.

“Pass” the updated substitution to the recursive call and collect the result of the recursive call.

Indicate that an abs has just been visited.

Pattern-match on app#f.

```
      beta_redex <- None;
      let orig_subst = subst in
      subst <- (v', arg')::subst;
      body'#accept self;
      subst <- orig_subst
    | None -> result := new_app f' arg'
end

class ['visitor] freevars_lambda (result : var_name list ref) =
object (self)
  inherit ['visitor] lambda_visit
  method visitVar var =
    result := var#v :: !result
  method visitAbs abs =
    (abs#body)#accept self;
    result := List.filter (fun v' -> v' <> abs#v) !result
  method visitApp app =
    app#arg#accept self; app#f#accept self
end

type lambda_visit_t = lambda_visit_t lambda_visit
type lambda_t = lambda_visit_t visitable

let eval1 (e : lambda_t) subst : lambda_t =
  let result = ref (new_var "") in
  e#accept (new eval_lambda subst result :> lambda_visit_t);
  !result
```

We use result as an accumulator.

Visitor for the language of λ-expressions.

This initial value will be ignored.

```
let freevars1 (e : lambda_t) =
  let result = ref [] in                                Initial value of the accumulator.
  e#accept (new freevars_lambda result);
  !result

let test1 =
  (new_app (new_abs "x" (new_var "x")) (new_var "y") :> lambda_t)
let e_test = eval1 test1 []
let fv_test = freevars1 test1

class ['visitor] num (i : int) =
object (self)
  method i = i
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitNum self
end
let new_num i = (new num i :> 'a visitable)

class virtual ['visitor] operation
  (arg1 : 'visitor visitable) (arg2 : 'visitor visitable) =
object (self)                                           Shared accessor methods.
  method arg1 = arg1
  method arg2 = arg2
end

class ['visitor] add arg1 arg2 =
object (self)
  inherit ['visitor] operation arg1 arg2
```

```
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitAdd self
end
let new_add arg1 arg2 = (new add arg1 arg2 :> 'a visitable)

class ['visitor] mult arg1 arg2 =
object (self)
  inherit ['visitor] operation arg1 arg2
  method accept : 'visitor -> unit =
    fun visitor -> visitor#visitMult self
end
let new_mult arg1 arg2 = (new mult arg1 arg2 :> 'a visitable)
```

The sub-language of arithmetic expressions.

```
class virtual ['visitor] expr_visit =
object
  method virtual visitNum : 'visitor num -> unit
  method virtual visitAdd : 'visitor add -> unit
  method virtual visitMult : 'visitor mult -> unit
end

class ['visitor] eval_expr
  (result : 'visitor visitable ref) =
object (self)
  inherit ['visitor] expr_visit
  val mutable num_redex : int option = None
```

The numeric result, if any.

```
  method visitNum num =
    num_redex <- Some num#i;
    result := (num :> 'visitor visitable)
```

```
  method private visitOperation new_e op e =
    (e#arg1)#accept self;
    let arg1' = !result and i1 = num_redex in
    (e#arg2)#accept self;
    let arg2' = !result and i2 = num_redex in
    match i1, i2 with
    | Some i1, Some i2 ->
      let res = op i1 i2 in
      num_redex <- Some res; result := new_num res
    | _ ->
      num_redex <- None;
      result := new_e arg1' arg2'
  method visitAdd add = self#visitOperation new_add ( + ) add
  method visitMult mult = self#visitOperation new_mult ( * ) mult
end
```

Flow-through class for computing free variables.

```
class ['visitor] freevars_expr (result : var_name list ref) =
object (self)
  inherit ['visitor] expr_visit
  method visitNum _ = ()
  method visitAdd add =
    add#arg1#accept self; add#arg2#accept self
  method visitMult mult =
    mult#arg1#accept self; mult#arg2#accept self
end
```

The language of arithmetic expressions – in this example without variables.

```
type expr_visit_t = expr_visit_t expr_visit
type expr_t = expr_visit_t visitable
```

```
let eval2 (e : expr_t) : expr_t =
  let result = ref (new_num 0) in
  e#accept (new eval_expr result);
  !result
```

This initial value will be ignored.

```
let test2 =
  (new_add (new_mult (new_num 3) (new_num 3)) (new_num 1) :> expr_t)
let e_test = eval2 test2
```

Combining the variants / constructors.

```
class virtual ['visitor] lexpr_visit =
object
  inherit ['visitor] lambda_visit
  inherit ['visitor] expr_visit
end
```

Combining the "pattern-matching branches".

```
class ['visitor] eval_lexpr subst result =
object
  inherit ['visitor] eval_expr result
  inherit ['visitor] eval_lambda subst result
end

class ['visitor] freevars_lexpr result =
object
  inherit ['visitor] freevars_expr result
  inherit ['visitor] freevars_lambda result
end
```

```
type lexpr_visit_t = lexpr_visit_t lexpr_visit
type lexpr_t = lexpr_visit_t visitable
```

The language combining λ-expressions and arithmetic expressions.

```
let eval3 (e : lexpr_t) subst : lexpr_t =
  let result = ref (new_num 0) in
  e#accept (new eval_lexpr subst result);
  !result

let freevars3 (e : lexpr_t) =
  let result = ref [] in
  e#accept (new freevars_lexpr result);
  !result

let test3 =
  (new_add (new_mult (new_num 3) (new_var "x")) (new_num 1) :> lexpr_t)
let e_test = eval3 test3 []
let fv_test = freevars3 test3
let old_e_test = eval3 (test2 :> lexpr_t) []
let old_fv_test = eval3 (test2 :> lexpr_t) []
```

- Polymorphic variants are to ordinary variants as objects are to records: both enable *open types* and subtyping, both allow different types to share the same components.
  - They are *dual* concepts in that if we replace "product" of records / objects by "sum" (see lecture 2), we get variants / polymorphic variants.
    Duality implies many behaviors are opposite.
- While object subtypes have more methods, polymorphic variant subtypes have less tags.
- The > sign means "these tags or more":

```
let l = [‘Int 3; ‘Float 4.];;
val l : [> ‘Float of float | ‘Int of int ] list = [‘Int 3; ‘Float 4.]
```

- The < sign means "these tags or less":

```
let is_positive = function
    | ‘Int   x -> Some (x > 0)
    | ‘Float x -> Some (x > 0.)
    | ‘Not_a_number -> None;;
val is_positive :
  [< ‘Float of float | ‘Int of int | ‘Not_a_number ] ->
    bool option = <fun>
```

- No sign means a closed type (similar to an object type without the ..)

- Both an upper and a lower bound are sometimes inferred, see https://realworldocaml.org/v1/en/html/variants.html

```
List.filter
  (fun x -> match is_positive x with None -> false | Some b -> b) l;;
- : [< `Float of float | `Int of int | `Not_a_number > `Float `Int ]
      list =
[`Int 3; `Float 4.]
```

- Because distinct polymorphic variant types can share the same tags, the solution to the Expression Problem is straightforward.
- Penalty points:
  - The need to "tie the recursive knot" separately both at the type level and the function level. At the function level, an η-expansion is required due to *value recursion* problem. At the type level, the type variable can be confusing.
  - There can be a slight time cost compared to the visitor pattern-based approach: additional dispatch at each level of type aggregation (i.e. merging sub-languages).

  Verdict: a flexible and concise solution, second-best place.

```
type var = [`Var of string]

let eval_var sub (`Var s as v : var) =
  try List.assoc s sub with Not_found -> v

type 'a lambda =
  [`Var of string | `Abs of string * 'a | `App of 'a * 'a]

let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n
```

```
let eval_lambda eval_rec subst : 'a lambda -> 'a = function
  | #var as v -> eval_var subst v
  | `App (l1, l2) ->
    let l2' = eval_rec subst l2 in
    (match eval_rec subst l1 with
    | `Abs (s, body) ->
      eval_rec [s, l2'] body
    | l1' -> `App (l1', l2'))
  | `Abs (s, l1) ->
    let s' = gensym () in
    `Abs (s', eval_rec ((s, `Var s')::subst) l1)
```

We could also leave the type open rather than closing it to `lambda`.

```
let freevars_lambda freevars_rec : 'a lambda -> 'b = function
  | `Var v -> [v]
  | `App (l1, l2) -> freevars_rec l1 @ freevars_rec l2
  | `Abs (s, l1) ->
    List.filter (fun v -> v <> s) (freevars_rec l1)

type lambda_t = lambda_t lambda

let rec eval1 subst e : lambda_t = eval_lambda eval1 subst e
let rec freevars1 (e : lambda_t) = freevars_lambda freevars1 e
```

```
let test1 = (`App (`Abs ("x", `Var "x"), `Var "y") :> lambda_t)
let e_test = eval1 [] test1
let fv_test = freevars1 test1

type 'a expr =
  [`Var of string | `Num of int | `Add of 'a * 'a | `Mult of 'a * 'a]

let map_expr (f : _ -> 'a) : 'a expr -> 'a = function
  | #var as v -> v
  | `Num _ as n -> n
  | `Add (e1, e2) -> `Add (f e1, f e2)
  | `Mult (e1, e2) -> `Mult (f e1, f e2)

let eval_expr eval_rec subst (e : 'a expr) : 'a =
  match map_expr (eval_rec subst) e with
  | #var as v -> eval_var subst v
  | `Add (`Num m, `Num n) -> `Num (m + n)
  | `Mult (`Num m, `Num n) -> `Num (m * n)
  | e -> e

let freevars_expr freevars_rec : 'a expr -> 'b = function
  | `Var v -> [v]
  | `Num _ -> []
```

Here and elsewhere, we could also factor-out the sub-language of variables.

```
  | `Add (e1, e2) | `Mult (e1, e2) -> freevars_rec e1 @ freevars_rec e2

type expr_t = expr_t expr

let rec eval2 subst e : expr_t = eval_expr eval2 subst e
let rec freevars2 (e : expr_t) = freevars_expr freevars2 e

let test2 = (`Add (`Mult (`Num 3, `Var "x"), `Num 1) : expr_t)
let e_test2 = eval2 ["x", `Num 2] test2
let fv_test2 = freevars2 test2

type 'a lexpr = ['a lambda | 'a expr]

let eval_lexpr eval_rec subst : 'a lexpr -> 'a = function
  | #lambda as x -> eval_lambda eval_rec subst x
  | #expr as x -> eval_expr eval_rec subst x

let freevars_lexpr freevars_rec : 'a lexpr -> 'b = function
  | #lambda as x -> freevars_lambda freevars_rec x
  | #expr as x -> freevars_expr freevars_rec x

type lexpr_t = lexpr_t lexpr
```

```
let rec eval3 subst e : lexpr_t = eval_lexpr eval3 subst e
let rec freevars3 (e : lexpr_t) = freevars_lexpr freevars3 e

let test3 =
  (`App (`Abs ("x", `Add (`Mult (`Num 3, `Var "x"), `Num 1)),
          `Num 2) : lexpr_t)
let e_test3 = eval3 [] test3
let fv_test3 = freevars3 test3
let e_old_test = eval3 [] (test2 :> lexpr_t)
let fv_old_test = freevars3 (test2 :> lexpr_t)
```

- Using recursive modules, we can clean-up the confusing or cluttering aspects of tying the recursive knots: type variables, recursive call arguments.
- We need *private types*, which for objects and polymorphic variants means *private rows*.
  - We can conceive of open row types, e.g. `[> ‘Int of int | ‘String of string]` as using a *row variable*, e.g. `’a`:

    ```
    [‘Int of int | ‘String of string | ’a]
    ```

    and then of private row types as abstracting the row variable:

    ```
    type t_row
    type t = [‘Int of int | ‘String of string | t_row]
    ```

    But the actual formalization of private row types is more complex.
- Penalty points:
  - We still need to tie the recursive knots for types, for example `private [> ’a lambda] as ’a`.
  - There can be slight time costs due to the use of functors and dispatch on merging of sub-languages.
- Verdict: a clean solution, best place.

```
type var = [`Var of string]

let eval_var subst (`Var s as v : var) =
  try List.assoc s subst with Not_found -> v

type 'a lambda =
  [`Var of string | `Abs of string * 'a | `App of 'a * 'a]

module type Eval =
sig type exp val eval : (string * exp) list -> exp -> exp end

module LF(X : Eval with type exp = private [> 'a lambda] as 'a) =
struct
  type exp = X.exp lambda

  let gensym =
    let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n

  let eval subst : exp -> X.exp = function
    | #var as v -> eval_var subst v
    | `App (l1, l2) ->
      let l2' = X.eval subst l2 in
      (match X.eval subst l1 with
```

```
      | `Abs (s, body) ->
        X.eval [s, l2'] body
      | l1' -> `App (l1', l2'))
    | `Abs (s, l1) ->
      let s' = gensym () in
      `Abs (s', X.eval ((s, `Var s')::subst) l1)
end
module rec Lambda : (Eval with type exp = Lambda.exp lambda) =
  LF(Lambda)

module type FreeVars =
sig type exp val freevars : exp -> string list end

module LFVF(X : FreeVars with type exp = private [> 'a lambda] as 'a) =
struct
  type exp = X.exp lambda

  let freevars : exp -> 'b = function
    | `Var v -> [v]
    | `App (l1, l2) -> X.freevars l1 @ X.freevars l2
    | `Abs (s, l1) ->
      List.filter (fun v -> v <> s) (X.freevars l1)
end
```

```
module rec LambdaFV : (FreeVars with type exp = LambdaFV.exp lambda) =
  LFVF(LambdaFV)

let test1 = (‘App (‘Abs ("x", ‘Var "x"), ‘Var "y") : Lambda.exp)
let e_test = Lambda.eval [] test1
let fv_test = LambdaFV.freevars test1

type ’a expr =
  [‘Var of string | ‘Num of int | ‘Add of ’a * ’a | ‘Mult of ’a * ’a]

module type Operations =
sig include Eval include FreeVars with type exp := exp end

module EF(X : Operations with type exp = private [> ’a expr] as ’a) =
struct
  type exp = X.exp expr

  let map_expr f = function
    | #var as v -> v
    | ‘Num _ as n -> n
    | ‘Add (e1, e2) -> ‘Add (f e1, f e2)
    | ‘Mult (e1, e2) -> ‘Mult (f e1, f e2)
```

```
  let eval subst (e : exp) : X.exp =
    match map_expr (X.eval subst) e with
    | #var as v -> eval_var subst v
    | `Add (`Num m, `Num n) -> `Num (m + n)
    | `Mult (`Num m, `Num n) -> `Num (m * n)
    | e -> e

  let freevars : exp -> 'b = function
    | `Var v -> [v]
    | `Num _ -> []
    | `Add (e1, e2) | `Mult (e1, e2) -> X.freevars e1 @ X.freevars e2
end
module rec Expr : (Operations with type exp = Expr.exp expr) =
  EF(Expr)

let test2 = (`Add (`Mult (`Num 3, `Var "x"), `Num 1) : Expr.exp)
let e_test2 = Expr.eval ["x", `Num 2] test2
let fvs_test2 = Expr.freevars test2

type 'a lexpr = ['a lambda | 'a expr]

module LEF(X : Operations with type exp = private [> 'a lexpr] as 'a) =
struct
```

```
  type exp = X.exp lexpr
  module LambdaX = LF(X)
  module LambdaFVX = LFVF(X)
  module ExprX = EF(X)

  let eval subst : exp -> X.exp = function
    | #LambdaX.exp as x -> LambdaX.eval subst x
    | #ExprX.exp as x -> ExprX.eval subst x

  let freevars : exp -> 'b = function
    | #lambda as x -> LambdaFVX.freevars x      Either of #lambda or #LambdaX.exp is fine.
    | #expr as x -> ExprX.freevars x            Either of #expr or #ExprX.exp is fine.
end
module rec LExpr : (Operations with type exp = LExpr.exp lexpr) =
  LEF(LExpr)

let test3 =
  (`App (`Abs ("x", `Add (`Mult (`Num 3, `Var "x"), `Num 1)),
         `Num 2) : LExpr.exp)
let e_test3 = LExpr.eval [] test3
let fv_test3 = LExpr.freevars test3
let e_old_test = LExpr.eval [] (test2 :> LExpr.exp)
let fv_old_test = LExpr.freevars (test2 :> LExpr.exp)
```

- We have done parsing using external languages OCamlLex and Menhir, now we will look at parsers written directly in OCaml.
- Language *combinators* are ways defining languages by composing definitions of smaller languages. For example, the combinators of the *Extended Backus-Naur Form* notation are:
  - concatenation: $S = A, B$ stands for $S = \{ab | a \in A, b \in b\}$,
  - alternation: $S = A | B$ stands for $S = \{a | a \in A \vee a \in B\}$,
  - option: $S = [A]$ stands for $S = \{\epsilon\} \cup A$, where ϵ is an empty string,
  - repetition: $S = \{A\}$ stands for $S = \{\epsilon\} \cup \{as | a \in A, s \in S\}$,
  - terminal string: $S ='' a''$ stands for $S = \{a\}$.
- Parsers implemented directly in a functional programming paradigm are functions from character streams to the parsed values. Algorithmically they are *recursive descent parsers*.
- *Parser combinators* approach builds parsers as *monad plus* values:
  - Bind: `val (>>=) : 'a parser -> ('a -> 'b parser) -> 'b parser`
    - `p >>= f` is a parser that first parses `p`, and makes the result available for parsing `f`.
  - Return: `val return : 'a -> 'a parser`
    - `return x` parses an empty string, symbolically $S = \{\epsilon\}$, and returns `x`.

- MZero: `val fail : 'a parser`
  - `fail` fails to parse anything, symbolically $S = \varnothing = \{\}$.
- MPlus: either `val <|> : 'a parser -> 'a parser -> 'a parser`,

  or `val <|> : 'a parser -> 'b parser -> ('a, 'b) choice parser`
  - `p <|> q` tries p, and if p succeeds, its result is returned, otherwise the parser q is used.

The only non-monad-plus operation that has to be built into the monad is some way to consume a single character from the input stream, for example:

- `val satisfy : (char -> bool) -> char parser`
  - `satisfy (fun c -> c = 'a')` consumes the character "a" from the input stream and returns it; if the input stream starts with a different character, this parser fails.

* Ordinary monadic recursive descent parsers **do not allow** *left-recursion*: if a cycle of calls not consuming any character can be entered when a parse failure should occur, the cycle will keep repeating.
  - For example, if we define numbers $N := D|ND$, where D stands for digits, then a stack of uses of the rule $N \rightarrow ND$ will build up when the next character is not a digit.
  - On the other hand, rules can share common prefixes.

- The parser monad is actually a composition of two monads:
  - the state monad for storing the stream of characters that remain to be parsed,
  - the backtracking monad for handling parse failures and ambiguities.

  Alternatively, one can split the state monad into a reader monad with the parsed string, and a state monad with the parsing position.
- Recall Lecture 8, especially slides 54-63.
- On my new OPAM installation of OCaml, I run the parsing example with:

  ```
  ocamlbuild Plugin1.cmxs -pp "camlp4o /home/lukstafi/.opam/4.02.1/lib/monad-custom/pa_monad.cmo"
  ```

  ```
  ocamlbuild Plugin2.cmxs -pp "camlp4o /home/lukstafi/.opam/4.02.1/lib/monad-custom/pa_monad.cmo"
  ```

  ```
  ocamlbuild PluginRun.native -lib dynlink -pp "camlp4o ~/.opam/4.02.1/lib/monad-custom/pa_monad.cmo" -- "(3*(6+1))" _build/Plugin1.cmxs _build/Plugin2.cmxs
  ```
- We experiment with a different approach to *monad-plus*. The merits of this approach (or lack thereof) is left as an exercise. *lazy-monad-plus*:

  ```
  val mplus : 'a monad -> 'a monad Lazy.t -> 'a monad
  ```

- Excerpts from `Monad.ml`. First an operation from `MonadPlusOps`.

```
let msum_map f l =
  List.fold_left
    (fun acc a -> mplus acc (lazy (f a))) mzero l
```

Folding left reversers the apparent order of composition, order from `l` is preserved.

- The implementation of the lazy-monad-plus.

```
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t
let rec ltake n = function
 | LCons (a, l) when n > 1 -> a::(ltake (n-1) (Lazy.force l))
 | LCons (a, l) when n = 1 -> [a]
 | _ -> []
let rec lappend l1 l2 =
  match l1 with LNil -> Lazy.force l2
  | LCons (hd, tl) -> LCons (hd, lazy (lappend (Lazy.force tl) l2))
let rec lconcat_map f = function
  | LNil -> LNil
  | LCons (a, l) -> lappend (f a) (lazy (lconcat_map f (Lazy.force l)))
module LListM = MonadPlus (struct
  type 'a t = 'a llist
  let bind a b = lconcat_map b a
  let return a = LCons (a, lazy LNil)
  let mzero = LNil
  let mplus = lappend
end)
```

Avoid forcing the tail if not needed.

- File Parsec.ml:

```
open Monad
module type PARSE = sig
  type 'a backtracking_monad                                    Name for the underlying monad-plus.
  type 'a parsing_state = int -> ('a * int) backtracking_monad       Processing state – position.
  type 'a t = string -> 'a parsing_state                             Reader for the parsed text.
  include MONAD_PLUS_OPS
  val (<|>) : 'a monad -> 'a monad Lazy.t -> 'a monad                    A synonym for mplus.
  val run : 'a monad -> 'a t
  val runT : 'a monad -> string -> int -> 'a backtracking_monad
  val satisfy : (char -> bool) -> char monad          Consume a character of the specified class.
  val end_of_text : unit monad                          Check for end of the processed text.
end
module ParseT (MP : MONAD_PLUS_OPS) :
  PARSE with type 'a backtracking_monad := 'a MP.monad =
struct
  type 'a backtracking_monad = 'a MP.monad
  type 'a parsing_state = int -> ('a * int) MP.monad
  module M = struct
    type 'a t = string -> 'a parsing_state
    let return a = fun s p -> MP.return (a, p)
    let bind m b = fun s p ->
      MP.bind (m s p) (fun (a, p') -> b a s p')
    let mzero = fun _ _ -> MP.mzero
    let mplus ma mb = fun s p ->
```

```
      MP.mplus (ma s p) (lazy (Lazy.force mb s p))
  end
  include M
  include MonadPlusOps(M)
  let (<|>) ma mb = mplus ma mb
  let runT m s p = MP.lift fst (m s p)
  let satisfy f s p =
    if p < String.length s && f s.[p]                    Consuming a character means accessing it
    then MP.return (s.[p], p + 1) else MP.mzero               and advancing the parsing position.
  let end_of_text s p =
    if p >= String.length s then MP.return ((), p) else MP.mzero
end
module type PARSE_OPS = sig
  include PARSE
  val many : 'a monad -> 'a list monad
  val opt : 'a monad -> 'a option monad
  val (?|) : 'a monad -> 'a option monad
  val seq : 'a monad -> 'b monad Lazy.t -> ('a * 'b) monad          Exercise: why laziness here?
  val (<*>) : 'a monad -> 'b monad Lazy.t -> ('a * 'b) monad                     Synonym for seq.
  val lowercase : char monad
  val uppercase : char monad
  val digit : char monad
  val alpha : char monad
  val alphanum : char monad
  val literal : string -> unit monad                      Consume characters of the given string.
  val (<<>) : string -> 'a monad -> 'a monad                        Prefix and postfix keywords.
  val (<>>) : 'a monad -> string -> 'a monad
end
```

```
module ParseOps (R : MONAD_PLUS_OPS)
  (P : PARSE with type 'a backtracking_monad := 'a R.monad) :
  PARSE_OPS with type 'a backtracking_monad := 'a R.monad =
struct
  include P
  let rec many p =
    (perform
        r <-- p; rs <-- many p; return (r::rs))
    ++ lazy (return [])
  let opt p = (p >>= (fun x -> return (Some x))) ++ lazy (return None)
  let (?|) p = opt p
  let seq p q = perform
      x <-- p; y <-- Lazy.force q; return (x, y)
  let (<*>) p q = seq p q
  let lowercase = satisfy (fun c -> c >= 'a' && c <= 'z')
  let uppercase = satisfy (fun c -> c >= 'A' && c <= 'Z')
  let digit = satisfy (fun c -> c >= '0' && c <= '9')
  let alpha = lowercase ++ lazy uppercase
  let alphanum = alpha ++ lazy digit
  let literal l =
    let rec loop pos =
      if pos = String.length l then return ()
      else satisfy (fun c -> c = l.[pos]) >>- loop (pos + 1) in
    loop 0
  let (<<>) bra p = literal bra >>- p
  let (<>>) p ket = p >>= (fun x -> literal ket >>- return x)
end
```

- File PluginBase.ml:

```
module ParseM =
  Parsec.ParseOps (Monad.LListM) (Parsec.ParseT (Monad.LListM))
open ParseM

let grammar_rules : (int monad -> int monad) list ref = ref []

let get_language () : int monad =
  let rec result =
    lazy
      (List.fold_left
         (fun acc lang -> acc <|> lazy (lang (Lazy.force result)))
          mzero !grammar_rules) in
  perform r <-- Lazy.force result; end_of_text; return r
```

Ensure we parse the whole text.

- File PluginRun.ml:

```
let load_plug fname : unit =
  let fname = Dynlink.adapt_filename fname in
  if Sys.file_exists fname then
    try Dynlink.loadfile fname
    with
    | (Dynlink.Error err) as e ->
      Printf.printf "\nERROR loading plugin: %s\n%!"
        (Dynlink.error_message err);
      raise e
    | e -> Printf.printf "\nUnknow error while loading plugin\n%!"
  else (
    Printf.printf "\nPlugin file %s does not exist\n%!" fname;
    exit (-1))

let () =
  for i = 2 to Array.length Sys.argv - 1 do
    load_plug Sys.argv.(i) done;
  let lang = PluginBase.get_language () in
  let result =
    Monad.LListM.run
      (PluginBase.ParseM.runT lang Sys.argv.(1) 0) in
  match Monad.ltake 1 result with
  | [] -> Printf.printf "\nParse error\n%!"
  | r::_ -> Printf.printf "\nResult: %d\n%!" r
```

Parser Combinators: Toy Example

- File `Plugin1.ml`:

```
open PluginBase.ParseM
let digit_of_char d = int_of_char d - int_of_char '0'
let number _ =
  let rec num =
```

Numbers: $N := DN|D$ where D is digits.

```
    lazy (  (perform
                d <-- digit;
                (n, b) <-- Lazy.force num;
                return (digit_of_char d * b + n, b * 10))
      <|> lazy (digit >>= (fun d -> return (digit_of_char d, 10)))) in
  Lazy.force num >>| fst
let addition lang =
```

Addition rule: $S \rightarrow (S + S)$.

Requiring a parenthesis (turns the rule into non-left-recursive.

```
  perform
    literal "("; n1 <-- lang; literal "+"; n2 <-- lang; literal ")";
    return (n1 + n2)
let () = PluginBase.(grammar_rules := number :: addition :: !grammar_rules)
```

- File `Plugin2.ml`:

```
open PluginBase.ParseM
let multiplication lang =
```

Multiplication rule: $S \rightarrow (S * S)$.

```
  perform
    literal "("; n1 <-- lang; literal "*"; n2 <-- lang; literal ")";
    return (n1 * n2)
let () = PluginBase.(grammar_rules := multiplication :: !grammar_rules)
```