User-Defined Types

Type Declarations

A new type name is defined globally. Unlike *let*, *type* is recursive by default, so the name being defined may appear in the *typedef*.

$$\textbf{type}\ \textit{name} = \textit{typedef}$$

Mutually-recursive types can be defined with *and*.

$$\begin{array}{ll} \textbf{type} & \textit{name}_1 = \textit{typedef}_1 \\ \textbf{and} & \textit{name}_2 = \textit{typedef}_2 \\ & \vdots \\ \textbf{and} & \textit{name}_n = \textit{typedef}_n \end{array}$$

Records

OCaml supports records much like C's *structs*.

```
# type base = { x : int; y : int; name : string };;
type base = { x : int; y : int; name : string; }

# let b0 = { x = 0; y = 0; name = "home" };;
val b0 : base = {x = 0; y = 0; name = "home"}
# let b1 = { b0 with x = 90; name = "first" };;
val b1 : base = {x = 90; y = 0; name = "first"}
# let b2 = { b1 with y = 90; name = "second" };;
val b2 : base = {x = 90; y = 90; name = "second"}

# b0.name;;
- : string = "home"

# let dist b1 b2 =
    let hyp x y = sqrt (float_of_int (x*x + y*y)) in
    hyp (b1.x - b2.x) (b1.y - b2.y);;
val dist : base -> base -> float = <fun>

# dist b0 b1;;
- : float = 90.
# dist b0 b2;;
- : float = 127.279220613578559
```

Algebraic Types/Tagged Unions/Sum-Product Types

Vaguely like C's *unions*, *enums*, or a class hierarchy: objects that can be one of a set of types. In compilers, great for trees and instructions.

```
# type seasons = Winter | Spring | Summer | Fall;;
type seasons = Winter | Spring | Summer | Fall

# let weather = function
    Winter -> "Too Cold"
  | Spring -> "Too Wet"
  | Summer -> "Too Hot"
  | Fall -> "Too Short";;
val weather : seasons -> string = <fun>

# weather Spring;;
- : string = "Too Wet"

# let year = [Winter; Spring; Summer; Fall] in
  List.map weather year;;
- : string list = ["Too Cold"; "Too Wet"; "Too Hot"; "Too Short"]
```

Simple Syntax Trees and an Interpreter

```
# type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr;;
type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr

# let rec eval = function
    Lit(x) -> x
  | Plus(e1, e2) -> (eval e1) + (eval e2)
  | Minus(e1, e2) -> (eval e1) - (eval e2)
  | Times(e1, e2) -> (eval e1) * (eval e2);;
val eval : expr -> int = <fun>

# eval (Lit(42));;
- : int = 42
# eval (Plus(Lit(17), Lit(25)));;
- : int = 42
# eval (Plus(Times(Lit(3), Lit(2)), Lit(1)));;
- : int = 7
```

Algebraic Type Rules

Each tag name must begin with a capital letter

```
# let bad1 = left | right;;
Syntax error
```

Tag names must be globally unique (required for type inference)

```
# type weekend = Sat | Sun;;
type weekend = Sat | Sun
# type days = Sun | Mon | Tue;;
type days = Sun | Mon | Tue
# function Sat -> "sat" | Sun -> "sun";;
This pattern matches values of type days
but is here used to match values of type weekend
```

Algebraic Types and Pattern Matching

The compiler warns about missing cases:

```
# type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr;;
type expr =
    Lit of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr

# let rec eval = function
    Lit(x) -> x
  | Plus(e1, e2) -> (eval e1) + (eval e2)
  | Minus(e1, e2) -> (eval e1) - (eval e2);;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Times (_, _)
val eval : expr -> int = <fun>
```

The *Option* Type: A Safe Null Pointer

Part of the always-loaded core library:

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

This is a polymorphic algebraic type: 'a is any type. *None* is like a null pointer; *Some* is a non-null pointer. The compiler requires *None* to be handled explicitly.

```
# let rec sum = function
    []          -> 0                                  (* base case *)
  | None::tl    -> sum tl  (* handle the "null pointer" case *)
  | Some(x)::tl -> x + sum tl;;                     (* normal case *)
val sum : int option list -> int = <fun>

# sum [None; Some(5); None; Some(37)];;
- : int = 42
```

Algebraic Types vs. Classes and Enums

	Algebraic Types	Classes	Enums
Choice of Types	fixed	extensible	fixed
Operations	extensible	fixed	extensible
Fields	ordered	named	none
Hidden fields	none	supported	none
Recursive	yes	yes	no
Inheritance	none	supported	none
Case splitting	simple	costly	simple

An algebraic type is best when the set of types rarely change but you often want to add additional functions. Classes are good in exactly the opposite case.

Modules and Compilation

Modules

Each source file is a module and everything is public.

foo.ml

```
(* Module Foo  *)

type t = { x : int ; y : int }
let sum c = c.x + c.y
```

bar.ml

```
(* The dot notation  *)

let v = { Foo.x = 1 ;
          Foo.y = 2 };;
print_int (Foo.sum v)

(* Create a short name  *)

module F = Foo;;
print_int (F.sum v)

(* Import every name from
   a module with "open" *)

open Foo;;
print_int (sum v)
```

To compile and run these,

```
$ ocamlc -c foo.ml
  (creates foo.cmi foo.cmo)
$ ocamlc -c bar.ml
  (creates bar.cmi bar.cmo)
$ ocamlc -o ex foo.cmo bar.cmo
$ ./ex
333
```

Separating Interface and Implementation

stack.mli

```
type 'a t

exception Empty

val create : unit -> 'a t
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
val top : 'a t -> 'a
val clear : 'a t -> unit
val copy : 'a t -> 'a t
val is_empty : 'a t -> bool
val length : 'a t -> int
val iter : ('a -> unit) ->
                 'a t -> unit
```

stack.ml

```
type 'a t =
   { mutable c : 'a list }
exception Empty

let create () = { c = [] }
let clear s = s.c <- []
let copy s = { c = s.c }
let push x s = s.c <- x :: s.c

let pop s =
  match s.c with
    hd::tl -> s.c <- tl; hd
  | []     -> raise Empty

let top s =
  match s.c with
    hd::_ -> hd
  | []    -> raise Empty

let is_empty s = (s.c = [])
let length s = List.length s.c
let iter f s = List.iter f s.c
```

A Complete Interpreter in Three Slides

The Scanner and AST

scanner.mll

```
{ open Parser }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
      | '+'                  { PLUS }
      | '-'                  { MINUS }
      | '*'                  { TIMES }
      | '/'                  { DIVIDE }
      | ['0'-'9']+ as lit    { LITERAL(int_of_string lit) }
      | eof                  { EOF }
```

ast.mli

```
type operator = Add | Sub | Mul | Div

type expr =
    Binop of expr * operator * expr
  | Lit of int
```

The Parser

parser.mly

```
%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE EOF
%token <int> LITERAL

%left PLUS MINUS
%left TIMES DIVIDE

%start expr
%type <Ast.expr> expr

%%

expr:
  expr PLUS   expr  { Binop($1, Add, $3) }
| expr MINUS  expr  { Binop($1, Sub, $3) }
| expr TIMES  expr  { Binop($1, Mul, $3) }
| expr DIVIDE expr  { Binop($1, Div, $3) }
| LITERAL           { Lit($1) }
```

The Interpeter

calc.ml

```
open Ast

let rec eval = function
    Lit(x) -> x
  | Binop(e1, op, e2) ->
      let v1 = eval e1 and v2 = eval e2 in
      match op with
        Add -> v1 + v2
      | Sub -> v1 - v2
      | Mul -> v1 * v2
      | Div -> v1 / v2

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let expr = Parser.expr Scanner.token lexbuf in
  let result = eval expr in
  print_endline (string_of_int result)
```

Compiling the Interpreter

```
$ ocamllex scanner.mll # create scanner.ml
8 states, 267 transitions, table size 1116 bytes
$ ocamlyacc parser.mly # create parser.ml and parser.mli
$ ocamlc -c ast.mli    # compile AST types
$ ocamlc -c parser.mli # compile parser types
$ ocamlc -c scanner.ml # compile the scanner
$ ocamlc -c parser.ml  # compile the parser
$ ocamlc -c calc.ml    # compile the interpreter
$ ocamlc -o calc parser.cmo scanner.cmo calc.cmo
$ ./calc
2 * 3 + 4 * 5
26
$
```

Compiling with *ocamlbuild*

```
$ ls
ast.mli  calc.ml  parser.mly  scanner.mll
$ ocamlbuild calc.native   # Build everything
Finished, 15 targets (0 cached) in 00:00:00.
$ ls
ast.mli _build calc.ml calc.native parser.mly scanner.mll
$ ./calc.native
2 * 3 + 4 * 5
Ctrl-D
26
$ ocamlbuild -clean # Remove _build and all .native
```

Exceptions; Directed Graphs

Exceptions

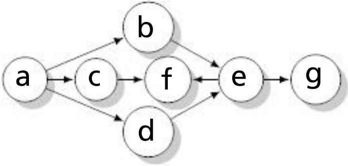
```
# 5 / 0;;
Exception: Division_by_zero.

# try
    5 / 0
  with Division_by_zero -> 42;;
- : int = 42

# exception My_exception;;
exception My_exception
# try
    if true then
       raise My_exception
    else 0
  with My_exception -> 42;;
- : int = 42
```

Exceptions

```
# exception Foo of string;;
exception Foo of string
# exception Bar of int * string;;
exception Bar of int * string

# let ex b =
  try
    if b then
      raise (Foo("hello"))
    else
      raise (Bar(42, " answer"))
  with Foo(s) -> "Foo: " ^ s
  | Bar(n, s) -> "Bar: " ^ string_of_int n ^ s;;
val ex : bool -> unit = <fun>

# ex true;;
- : string = "Foo: hello"
# ex false;;
- : string = "Bar: 42 answer"
```

Application: Directed Graphs

```
let edges = [
  ("a", "b"); ("a", "c");
  ("a", "d"); ("b", "e");
  ("c", "f"); ("d", "e");
  ("e", "f"); ("e", "g") ]

let rec successors n = function
    []                 -> []
  | (s, t) :: edges ->
      if s = n then
         t :: successors n edges
      else
         successors n edges
```




```
# successors "a" edges;;
- : string list = ["b"; "c"; "d"]

# successors "b" edges;;
- : string list = ["e"]
```

More Functional Successors

```
let rec successors n = function
    []                -> []
  | (s, t) :: edges ->
      if s = n then
         t :: successors n edges
      else
         successors n edges
```

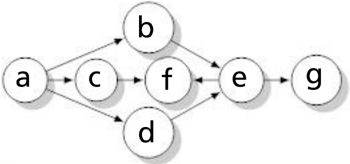
Our first example is imperative: performs "search a list," which is more precisely expressed using the library function `List.filter`:

```
let successors n edges =
    let matching (s,_) = s = n in
    List.map snd (List.filter matching edges)
```

This uses the built-in `snd` function, which is defined as

```
let snd (_,x) = x
```

Depth-First Search

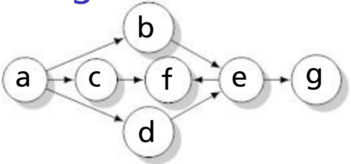


frontier nodes

```
let rec dfs edges visited = function
  []        -> List.rev visited
| n::nodes ->
  if List.mem n visited then
    dfs edges visited nodes
  else
    dfs edges (n::visited) ((successors n edges) @ nodes)
```

```
# dfs edges [] ["a"];;
- : string list = ["a"; "b"; "e"; "f"; "g"; "c"; "d"]
# dfs edges [] ["e"];;
- : string list = ["e"; "f"; "g"]
# dfs edges [] ["d"];;
- : string list = ["d"; "e"; "f"; "g"]
```

Topological Sort



Remember the visitor at the end.

```
let rec tsort edges visited = function
  []        -> visited
| n::nodes ->
  let visited' = if List.mem n visited then visited
                 else n :: tsort edges visited (successors n edges)
  in tsort edges visited' nodes;;
```

```
# tsort edges [] ["a"];;
- : string list = ["a"; "d"; "c"; "b"; "e"; "g"; "f"]

# let cycle = [ ("a", "b"); ("b", "c"); ("c", "a") ];;
val cycle : (string * string) list = [("a", "b"); ...]
# tsort cycle [] ["a"];;
Stack overflow during evaluation (looping recursion?).
```

Better Topological Sort

```
exception Cyclic of string

let tsort edges seed =
  let rec sort path visited = function
      []         -> visited
    | n::nodes ->
      if List.mem n path then raise (Cyclic n) else
      let v' = if List.mem n visited then visited else
               n :: sort (n::path) visited (successors n edges)
      in sort path v' nodes
  in
  sort [] [] [seed]
```

```
# tsort edges "a";;
- : string list = ["a"; "d"; "c"; "b"; "e"; "g"; "f"]

# tsort edges "d";;
- : string list = ["d"; "e"; "g"; "f"]

# tsort cycle "a";;
Exception: Cyclic "a".
```

Standard Library Modules

Maps

Balanced trees for implementing dictionaries. Ask for a map with a specific kind of key; values are polymorphic.

```
# module StringMap = Map.Make(String);;
module StringMap :
  sig
    type key = String.t
    type 'a t = 'a Map.Make(String).t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val compare : ('a -> 'a -> int) -> 'a t -> 'a t -> int
    val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  end
```

Maps

```
# let mymap = StringMap.empty;;            (* Create empty map *)
val mymap : 'a StringMap.t = <abstr>

# let mymap = StringMap.add "Douglas" 42 mymap;; (* Add pair *)
val mymap : int StringMap.t = <abstr>

# StringMap.mem "foo" mymap;;               (* Is "foo" there? *)
- : bool = false
# StringMap.mem "Douglas" mymap;;       (*Is "Douglas" there? *)
- : bool = true

# StringMap.find "Douglas" mymap;;                (* Get value *)
- : int = 42

# let mymap = StringMap.add "Adams" 17 mymap;;
val mymap : int StringMap.t = <abstr>

# StringMap.find "Adams" mymap;;
- : int = 17
# StringMap.find "Douglas" mymap;;
- : int = 42
# StringMap.find "Slarti" mymap;;
Exception: Not_found.
```

Maps

- Fully functional: *Map.add* takes a key, a value, and a map and returns a new map that also includes the given key/value pair.
- Needs a totally ordered key type. *Pervasives.compare* usually does the job (returns -1, 0, or 1); you may supply your own.

```
module StringMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)
```

- Uses balanced trees, so searching and insertion is $O(\log n)$.

Depth-First Search Revisited

Previous version

```
let rec dfs edges visited = function
  []       -> List.rev visited
| n::nodes ->
  if List.mem n visited then
    dfs edges visited nodes
  else
    dfs edges (n::visited) ((successors n edges) @ nodes)
```

was not very efficient, but good enough for small graphs.

Would like faster *visited* test and *successors* query.

Depth-First Search Revisited

Second version:

- use a Map to hold a list of successors for each node
- use a Set (valueless Map) to remember of visited nodes

```
module StringMap = Map.Make(String)
module StringSet = Set.Make(String)
```

Depth-First Search Revisited

```
let top_sort_map edges =
  (* Create an empty successor list for each node *)
  let succs = List.fold_left
      (fun map (s,d) ->
        StringMap.add d [] (StringMap.add s [] map)
      ) StringMap.empty edges
  in
  (* Build the successor list for each source node *)
  let succs = List.fold_left
      (fun succs (s, d) ->
        let ss = StringMap.find s succs
        in StringMap.add s (d::ss) succs) succs edges
  in
  (*  Visit recursively, storing each node after visiting successors*)
  let rec visit (order, visited) n =
    if StringSet.mem n visited then
      (order, visited)
    else
      let (order, visited) = List.fold_left
          visit (order, StringSet.add n visited)
          (StringMap.find n succs)
      in (n::order, visited)
  in
   (* Visit the source of each edge *)
  fst (List.fold_left visit ([], StringSet.empty) (List.map fst edges))
```

Imperative Features

```
# 0 ; 42;;                              (* ";" means sequencing *)
Warning S: this expression should have type unit.
- : int = 42

# ignore 0 ; 42;;            (* ignore is a function: 'a -> unit *)
- : int = 42
# () ; 42;;                 (* () is the literal for the unit type *)
- : int = 42

# print_endline "Hello World!";;        (* Print; result is unit *)
Hello World!
- : unit = ()
# print_string "Hello " ; print_endline "World!";;
Hello World!
- : unit = ()

# print_int 42 ; print_newline ();;
42
- : unit = ()
# print_endline ("Hello " ^ string_of_int 42 ^ " world!");;
Hello 42 world!
- : unit = ()
```

Hash Tables

```
# module StringHash = Hashtbl.Make(struct
    type t = string                          (* type of keys *)
    let equal x y = x = y        (* use structural comparison *)
    let hash = Hashtbl.hash          (* generic hash function *)
  end);;
module StringHash :
  sig
    type key = string
    type 'a t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val length : 'a t -> int
  end
```

Hash Tables

```
# let hash = StringHash.create 17;;  (* initial size estimate *)
val hash : '_a StringHash.t = <abstr>

# StringHash.add hash "Douglas" 42;; (* modify the hash table *)
- : unit = ()

# StringHash.mem hash "foo";;              (* is "foo" there? *)
- : bool = false
# StringHash.mem hash "Douglas";;      (* is "Douglas" there? *)
- : bool = true

# StringHash.find hash "Douglas";;                (* Get value *)
- : int = 42

# StringHash.add hash "Adams" 17;;   (* Add another key/value *)
- : unit = ()

# StringHash.find hash "Adams";;
- : int = 17
# StringHash.find hash "Douglas";;
- : int = 42
# StringHash.find hash "Slarti";;
Exception: Not_found.
```

Arrays

```
# let a = [| 42; 17; 19 |];;                  (* Array literal *)
val a : int array = [|42; 17; 19|]
# let aa = Array.make 5 0;;                 (* Fill a new array *)
val aa : int array = [|0; 0; 0; 0; 0|]

# a.(0);;                                     (* Random access *)
- : int = 42
# a.(2);;
- : int = 19
# a.(3);;
Exception: Invalid_argument "index out of bounds".

# a.(2) <- 20;;                          (* Arrays are mutable! *)
- : unit = ()
# a;;
- : int array = [|42; 17; 20|]

# let l = [24; 32; 17];;
val l : int list = [24; 32; 17]
# let b = Array.of_list l;;               (* Array from a list *)
val b : int array = [|24; 32; 17|]

# let c = Array.append a b;;                  (* Concatenation *)
val c : int array = [|42; 17; 20; 24; 32; 17|]
```

Arrays vs. Lists

	Arrays	Lists
Random access	$O(1)$	$O(n)$
Appending	$O(n)$	$O(1)$
Mutable	Yes	No

prepend

Useful pattern: first collect data of unknown length in a list then convert it to an array with *Array.of_list* for random queries.

DFS with Arrays

Second version used a lot of *mem*, *find*, and *add* calls on the string map, each $O(\log n)$. (Total: O(nlogn) at least.) Can we do better?

Solution: use arrays to hold adjacency lists and track visiting information.

Basic idea: number the nodes, build adjacency lists with numbers, use an array for tracking visits, then transform back to list of node names.

DFS with Arrays 1/2

```
let top_sort_array edges =
  (* Assign a number to each node  *)
  let map, nodecount =
    List.fold_left
      (fun nodemap (s, d) ->
        let addnode node (map, n) =
          if StringMap.mem node map then (map, n)
          else (StringMap.add node n map, n+1)
        in
        addnode d (addnode s nodemap)
      ) (StringMap.empty, 0) edges
  in

  let successors = Array.make nodecount [] in
  let name = Array.make nodecount "" in

  (* Build adjacency lists and remember the name of each node *)
  List.iter
    (fun (s, d) ->
      let ss = StringMap.find s map in
      let dd = StringMap.find d map in
      successors.(ss) <- dd :: successors.(ss);
      name.(ss) <- s;
      name.(dd) <- d;
    ) edges;
```

DFS with Arrays 2/2

```
(* Visited flags for each node *)
let visited = Array.make nodecount false in

(* Visit each of our successors if we haven't done so yet*)
(* then record the node *)
let rec visit order n =
  if visited.(n) then order
  else (
    visited.(n) <- true;
    n :: (List.fold_left visit order successors.(n))
   )
in

(* Compute the topological order*)
let order = visit [] 0 in

(* Map node numbers back to node names*)
List.map (fun n -> name.(n)) order
```