Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- We will look at another example of the **CPS Transformation** that we saw last week
- Then, taking a step back—how would we actually **automate** transforming programs like this?
- We need a way to **represent** the syntax of our language that allows us to (1) **construct** a representation of a new (transformed) program, and (2) **match** over the syntax of the original
- We've seen something like this for lists—if we generalize, we get **datatypes**
- We'll cover **many kinds** of datatypes

Objectives for Today

- We will look at another example of the **CPS Transformation** that we saw last week
- Then, taking a step back—how would we actually **automate** transforming programs like this?
- We need a way to **represent** the syntax of our language that allows us to (1) **construct** a representation of a new (transformed) program, and (2) **match** over the syntax of the original
- We've seen something like this for lists—if we generalize, we get **datatypes**
- We'll cover **many kinds** of datatypes

Objectives for Today

- We will look at another example of the **CPS Transformation** that we saw last week
- Then, taking a step back—how would we actually **automate** transforming programs like this?
- We need a way to **represent** the syntax of our language that allows us to (1) **construct** a representation of a new (transformed) program, and (2) **match** over the syntax of the original
- We've seen something like this for lists—if we generalize, we get **datatypes**
- We'll cover **many kinds** of datatypes

Please post questions on Piazza!

CPS Transformation Example

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
   true
  else
   mem (y, xs)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

            k false (* rule 2 *)



         k true (* rule 2 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

            k false (* rule 2 *)



        k true (* rule 2 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

      k false (* rule 2 *)

    k true (* rule 2 *)

    memk (y, xs) k (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

        k false (* rule 2 *)


    k true (* rule 2 *)

    memk (y, xs) k (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

           k false (* rule 2 *)
                  (* rule 4 *)
    eqk (x, y) (fun b ->    b
      k true (* rule 2 *)

      memk (y, xs) k) (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

             k false (* rule 2 *)
                 (* rule 4 *)
     eqk (x, y) (fun b ->    b
       k true (* rule 2 *)

       memk (y, xs) k) (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

          k false (* rule 2 *)
              (* rule 4 *)
    eqk (x, y) (fun b -> if b then
      k true (* rule 2 *)
    else
      memk (y, xs) k) (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)

          k false (* rule 2 *)
                (* rule 4 *)
    eqk (x, y) (fun b -> if b then
      k true (* rule 2 *)
    else
      memk (y, xs) k) (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)
match lst with
| [ ] -> k false (* rule 2 *)
| x :: xs ->  (* rule 4 *)
    eqk (x, y) (fun b -> if b then
      k true (* rule 2 *)
    else
      memk (y, xs) k) (* rule 3 *)
```

CPS Example: List Membership

Before:

```
let rec mem (y, lst) =
match lst with
| [ ] -> false
| x :: xs ->
  if (x = y) then
    true
  else
    mem (y, xs)
```

After:

```
let rec memk (y, lst) k = (* rule 1 *)
match lst with
| [ ] -> k false (* rule 2 *)
| x :: xs ->  (* rule 4 *)
    eqk (x, y) (fun b -> if b then
      k true (* rule 2 *)
    else
      memk (y, xs) k) (* rule 3 *)
```

Please post questions on Piazza!

How to implement automatically in compiler, rather than by hand?

How do we even represent the syntax of our language, and map over it to transform programs?

Datatypes

OCaml Datatype You've Seen: lists

- Frequently used lists in recursive program
- Matched over two structural cases
  - [ ] - the empty list
  - (x :: xs) a non-empty list
- Covers all possible lists
- type 'a list = [ ] | (::) of 'a * 'a list
  - Not quite legitimate declaration because of special syntax

OCaml Datatype You've Seen: lists

- Frequently used lists in recursive program
- Matched over two structural cases
  - [ ] - the empty list
  - (x :: xs) a non-empty list
- Covers all possible lists
- type ‘a list = [ ] | (::) of ‘a * ‘a list
  - Not quite legitimate declaration because of special syntax

OCaml Datatypes in General

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *pattern matching* (alt. *destruction* or, with caveats, *elimination*)

OCaml Datatypes in General

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *pattern matching* (alt. *destruction* or, with some extra machinery, *elimination*)

OCaml Datatypes in General

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *pattern matching* (alt. *destruction* or, with some extra machinery, *elimination*)

Datatypes in General

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *case analysis* (alt. *destruction* or, with some extra machinery, *induction*)

OCaml Datatypes in General

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *pattern matching* (alt. *destruction* or, with some extra machinery, *elimination*)

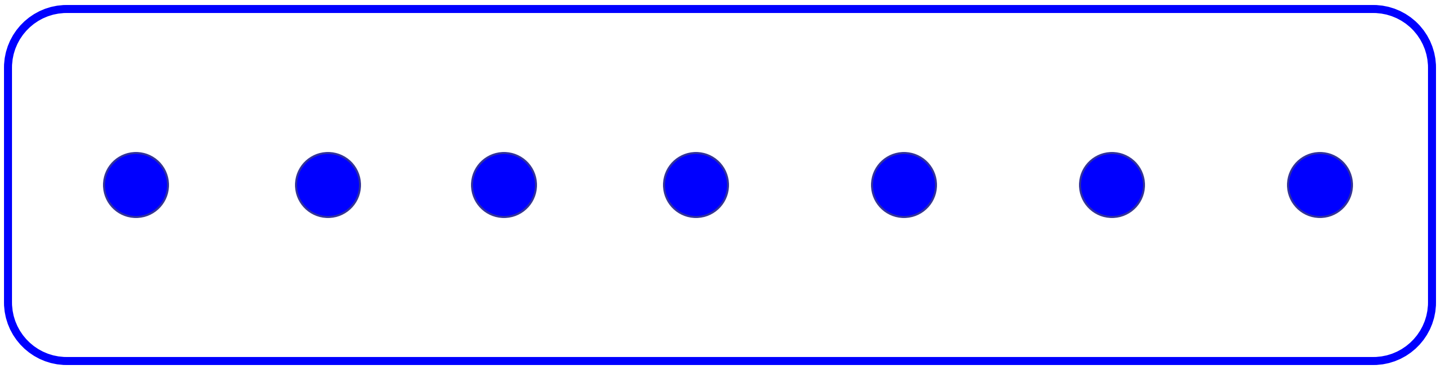
Enumeration Types

OCaml **Variants**

- type *name* = C_1 [of *ty_1*] | . . . | C_n [of *ty_n*]
- Introduce a type called *name*
- (fun x -> C_i x) : *ty_1* -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all *pattern matching* (alt. *destruction* or, with some extra machinery, *elimination*)

Enumeration Types as Variants

An enumeration type is a collection of distinct values



In C and Ocaml they have an order structure; order by order of input

Enumeration Types as Variants

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

Functions over Enumerations

```
# let day_after day =
    match day with
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | Friday -> Saturday
    | Saturday -> Sunday
    | Sunday -> Monday;;
val day_after : weekday -> weekday = <fun>
```

Functions over Enumerations

```
# let rec days_later n day =
    match n with
    | 0 -> day
    | _ ->
      if n > 0 then
        day_after (days_later (n - 1) day)
      else
        days_later (n + 7) day;;
val days_later : int -> weekday -> weekday = <fun>
```

Functions over Enumerations

```
# let rec days_later n day =
    match n with
    | 0 -> day
    | _ ->
      if n > 0 then
        day_after (days_later (n - 1) day)
      else
        days_later (n + 7) day;;
val days_later : int -> weekday -> weekday = <fun>
```

Functions over Enumerations

```
# let rec days_later n day =
    match n with
    | 0 -> day
    | _ ->
      if n > 0 then
        day_after (days_later (n - 1) day)
      else
        days_later (n + 7) day;;
val days_later : int -> weekday -> weekday = <fun>
```

Functions over Enumerations

```
# let rec days_later n day =
    match n with
    | 0 -> day
    | _ ->
      if n > 0 then
        day_after (days_later (n - 1) day)
      else
        days_later (n + 7) day;;
val days_later : int -> weekday -> weekday = <fun>
```

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
```

Your turn!

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
  match day with
```

Weekend days?

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
  match day with
  | _ -> true
```

In a better world …

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
  match day with
  | Saturday -> true
  | Sunday -> true
```

Other days?

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
  match day with
  | Saturday -> true
  | Sunday -> true
  | Monday -> false
  | Tuesday -> false …
```

More concisely?

Problem:

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

- Write function is_weekend : weekday -> bool

```
let is_weekend day =
  match day with
  | Saturday -> true
  | Sunday -> true
  | _ -> false
```

Yay

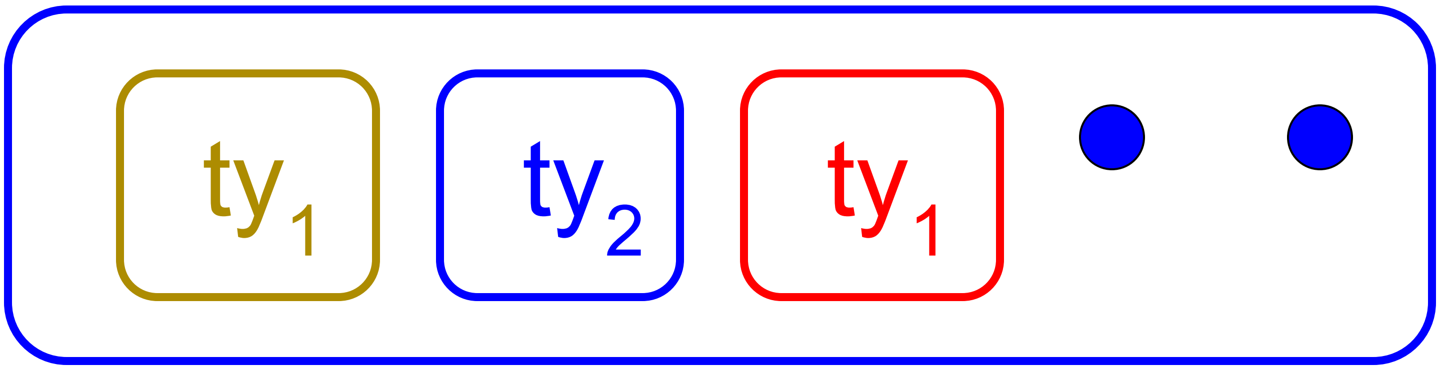
Enumeration Types in Languages!

```
# (* Binary operators *)
  type bin_op = IntPlusOp |  IntMinusOp
     |  EqOp | CommaOp | ConsOp

# (* Unary operators *)
  type mon_op = HdOp | TlOp | FstOp | SndOp
```

Disjoint Union Types

Disjoint Union Types as Variants

- Disjoint union of types, with some possibly occurring more than once



- We can also add in some new singleton elements

Disjoint Union Types

```
(* Different forms of identification *)
type id = DriversLicense of int
  | SocialSecurity of int | Name of string

let check_id id =
  match id with
  | DriversLicense num ->
      not (List.mem num [13570; 99999])
  | SocialSecurity num -> num < 900000000
  | Name str -> not (str = "John Doe")
```

Disjoint Union Types

```
(* Different forms of identification *)
type id = DriversLicense of int
  | SocialSecurity of int | Name of string

let check_id id =
  match id with
  | DriversLicense num ->
      not (List.mem num [13570; 99999])
  | SocialSecurity num -> num < 900000000
  | Name str -> not (str = "John Doe")
```

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

Your turn!

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

type currency =

How many constructors?

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

```
type currency =
 |                    (* US *)
 |                    (* UK *)
 |                  (* Europe *)
 |                 (* Japan *)
```

What currencies?

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

```
type currency =
 | Dollar        (* US *)
 | Pound         (* UK *)
 | Euro          (* Europe *)
 | Yen           (* Japan *)
```

How to store values?

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

```
type currency =
 | Dollar of int (* US *)
 | Pound of int (* UK *)
 | Euro of int (* Europe *)
 | Yen of int (* Japan *)
```

Disjoint Unions in Languages!

```
type const =
  BoolConst of bool
| IntConst of int
| FloatConst of float
| StringConst of string
| NilConst
| UnitConst
```

Disjoint Unions in Languages!

```
type const =
  BoolConst of bool
| IntConst of int
| FloatConst of float
| StringConst of string
| NilConst
| UnitConst
```

- How to represent 7 as a const?

Disjoint Unions in Languages!

```
type const =
  BoolConst of bool
| IntConst of int
| FloatConst of float
| StringConst of string
| NilConst
| UnitConst
```

- How to represent 7 as a const?
- Answer: IntConst 7

Please post questions on Piazza!

Polymorphic Datatypes

Polymorphism in Variants

- Variants can also be **polymorphic**
- For example, the type 'a option gives us something to represent non-existence or failure

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

- Used to encode partial functions
- Often can replace the raising of an exception

Polymorphism in Variants

- Variants can also be **polymorphic**
- For example, the type 'a option gives us something to represent non-existence or failure

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

- Used to encode partial functions
- Often can replace the raising of an exception

Polymorphism in Variants

- Variants can also be **polymorphic**
- For example, the type 'a option gives us something to represent non-existence or failure

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

- Used to encode partial functions
- Often can replace the raising of an exception

Polymorphism in Variants

- Variants can also be **polymorphic**
- For example, the type 'a option gives us something to represent non-existence or failure

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

- Used to encode partial functions
- Often can replace the raising of an exception

Functions producing option

```
# let rec first p list =
    match list with
    | [ ] -> None
    | (x :: xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
# first (fun x -> x > 3) [1; 3; 4; 2; 5];;
- : int option = Some 4
# first (fun x -> x > 5) [1; 3; 4; 2; 5];;
- : int option = None
```

Functions producing option

```
# let rec first p list =
    match list with
    | [ ] -> None
    | (x :: xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
# first (fun x -> x > 3) [1; 3; 4; 2; 5];;
- : int option = Some 4
# first (fun x -> x > 5) [1; 3; 4; 2; 5];;
- : int option = None
```

Functions producing option

```
# let rec first p list =
    match list with
    | [ ] -> None
    | (x :: xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
# first (fun x -> x > 3) [1; 3; 4; 2; 5];;
- : int option = Some 4
# first (fun x -> x > 5) [1; 3; 4; 2; 5];;
- : int option = None
```

Functions over option

```
# let result_ok r =
   match r with
   | None -> false
   | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1; 3; 4; 2; 5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1; 3; 4; 2; 5]);;
- : bool = false
```

Functions over option

```
# let result_ok r =
   match r with
   | None -> false
   | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1; 3; 4; 2; 5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1; 3; 4; 2; 5]);;
- : bool = false
```

Functions over option

```
# let result_ok r =
   match r with
   | None -> false
   | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1; 3; 4; 2; 5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1; 3; 4; 2; 5]);;
- : bool = false
```

Problem

- Write a hd on lists that doesn't raise an exception and works at all types of lists.

Your turn!

Problem

- Write a hd on lists that doesn't raise an exception and works at all types of lists.

```
let hd list =
  match list with
```

Nil case?

Problem

- Write a hd on lists that doesn't raise an exception and works at all types of lists.

```
let hd list =
  match list with
  | [] -> None
```

Cons case?

Problem

- Write a hd on lists that doesn't raise an exception and works at all types of lists.

```
let hd list =
  match list with
  | [] -> None
  | (x :: xs) -> Some x
```

Mapping over Variants

```
# let optionMap f opt =
    match opt with
    | None -> None
    | Some x -> Some (f x);;
val optionMap :
  ('a -> 'b) -> 'a option -> 'b option = <fun>
# optionMap
     (fun x -> x - 2)
     (first (fun x -> x > 3) [1; 3; 4; 2; 5]);;
-  : int option = Some 2
```

Mapping over Variants

```
# let optionMap f opt =
    match opt with
    | None -> None
    | Some x -> Some (f x);;
val optionMap :
  ('a -> 'b) -> 'a option -> 'b option = <fun>
# optionMap
     (fun x -> x - 2)
     (first (fun x -> x > 3) [1; 3; 4; 2; 5]);;
- : int option = Some 2
```

Folding over Variants

```
# let optionFold someFun noneVal opt =
    match opt with
    | None -> noneVal
    | Some x -> someFun x;;
val optionFold :
  ('a -> 'b) -> 'b -> 'a option -> 'b = <fun>
# let optionMap f opt =
    optionFold (fun x -> Some (f x)) None opt;;
val optionMap :
  ('a -> 'b) -> 'a option -> 'b option = <fun>
```

Folding over Variants

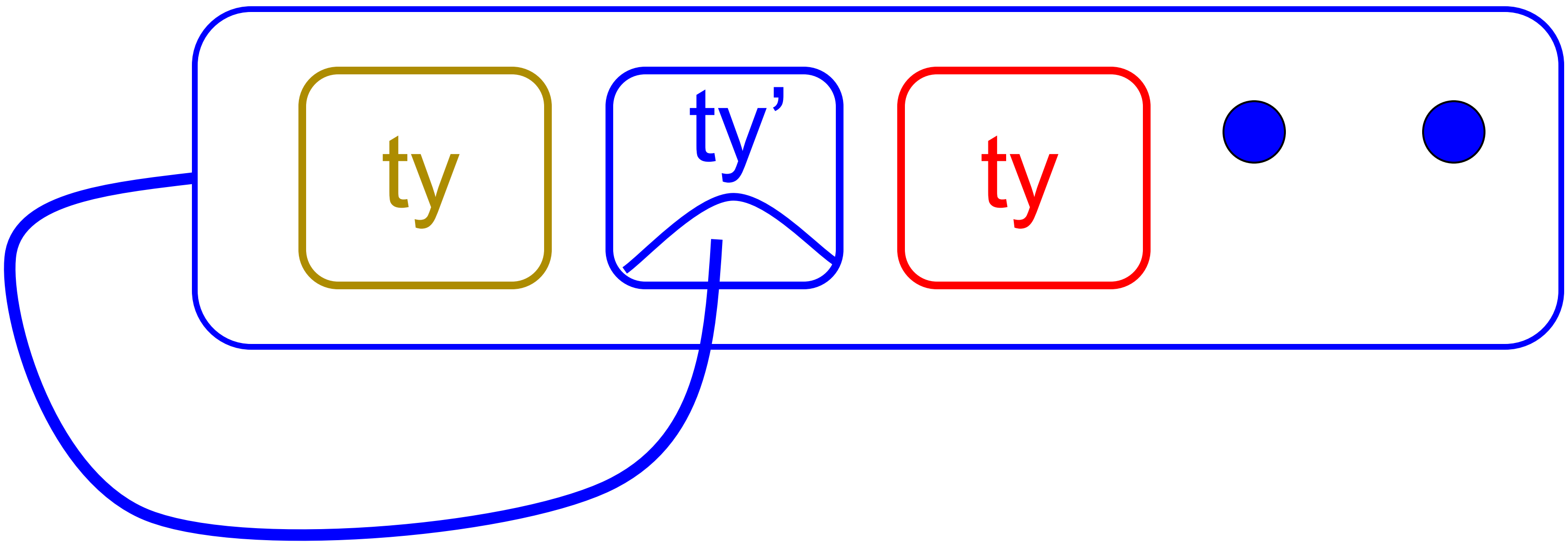
```
# let optionFold someFun noneVal opt =
    match opt with
    | None -> noneVal
    | Some x -> someFun x;;
val optionFold :
  ('a -> 'b) -> 'b -> 'a option -> 'b = <fun>
# let optionMap f opt =
    optionFold (fun x -> Some (f x)) None opt;;
val optionMap :
  ('a -> 'b) -> 'a option -> 'b option = <fun>
```

Please post questions on Piazza!

Preview: Recursive Datatypes

Recursive Types as Variants

- The type being defined may be a component of itself



Preview

Recursive Data Types

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

Preview

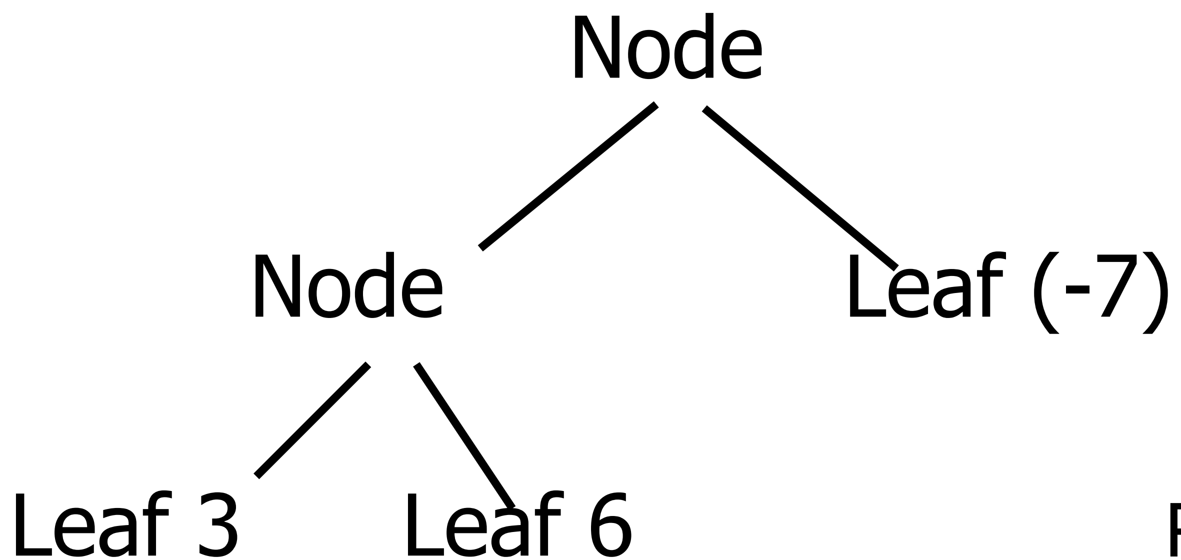
Recursive Data Types

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

Preview

Recursive Data Types

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Preview

Recursive Data Types in Languages!

```
# type exp =
    | VarExp of string
    | ConstExp of const
    | MonOpAppExp of mon_op * exp
    | BinOpAppExp of bin_op * exp * exp
    | IfExp of exp* exp * exp
    | AppExp of exp * exp
    | FunExp of string * exp
```

Preview

✓ How do we even represent the syntax of our language, and map over it to transform programs?

How to implement automatically in compiler, rather than by hand?

Please post questions on Piazza!

Takeaways

- **Variants** let us represent custom **datatypes**
  - Can be **polymorphic**
  - Can be **recursive**
  - Can represent **lists** and **trees**
  - Can represent **language syntax**!
- Can do **two things** with them:
  - **construct**
  - **destruct** (match, eliminate)
- Can write **program transformations**, **interpreters**, and **compilers** this way :)

Next Class

- **I will be back!** Lecture will happen in person
- **EC1** is due, if interested **(extra credit)**
- **WA3XC** also due, if interested **(extra credit)**
- **MP4** will be due next Tuesday
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help