Unions and Products

Concepts of Programming Languages
Lecture 7

CAS CS 320

Announcements

» Thanks for providing feedback on the course!

» Lectures will be more **interactive:** I will be asking more questions; we will do derivations and programming together

» More **examples**! Each concept will be followed by an example on typing, semantic rules, and programming. More practice via **homework.**

Practice Problem 1 (Tail Recursive)

Implement the function

reverse : 'a list -> 'a list

Practice Problem 2 (Tail Recursive)

Implement the function ***double*** *where* ***double l*** *doubles every element of the list* ***l***

Homework (Tail Recursive)

Implement the function

delete_every_other : ‘a list -> ‘a list

Outline

» Discuss the use of **tuples** and **records** for creating collections of data

» Introduce **algebraic data types (ADTs)** for creating data with given "shapes"

» Cover **parametric** and **recursive** ADTs for more general data structures

Products

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are **ordered unlabeled** fixed-length heterogeneous collections of data

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are **ordered unlabeled** fixed-length heterogeneous collections of data

(I expect that these are familiar)

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are **ordered unlabeled** fixed-length heterogeneous collections of data

(I expect that these are familiar)

These are useful for returning multiple values from a function

Pattern Matching on Tuples

```
let dist (p1 : float * float) (p2 : float * float) : float =
  match p1, p2 with
  | (x1, y1), (x2, y2) ->
      let x = x1 -. x2 in
      let y = y1 -. y2 in
      sqrt ((x *. x) +. (y *. y))
```

There are no accessors for tuples

Instead we can use **pattern matching**

Pattern Matching in General

```
match e with
| p -> o
| ...
```

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* <u>any</u> piece of data

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* <u>any</u> piece of data

We *match* on an expression e, and check if the value of e *matches* with the pattern p

Note: Patterns are not Expressions

```
<expr> ::= match <expr> with
           | <pattern> -> <expr>
           | <pattern> -> <expr>
           | ....
```

Patterns are similar to expressions, but with some key differences

They can be wildcards, they can be variables, there's a lot of possibilities (link)

Advanced Pattern Matching

```
let dist ((x1, y1) : float * float) ((x2, y2) : float * float) : float =
  let x = x1 -. x2 in
  let y = y1 -. y2 in
  sqrt ((x *. x) +. (y *. y))

let dist (p1 : float * float) (p2 : float * float) : float =
  let (x1, y1) = p1 in
  let (x2, y2) = p2 in
  let x = x1 -. x2 in
  let y = y1 -. y2 in
  sqrt ((x *. x) +. (y *. y))
```

Pattern matching can also be done implicitly in let-expressions and function arguments!

And we can do all this formally!

Tuples (Syntax Rule)

If $e_1, \ldots, e_n$ are well-formed expressions, then

$$(\; e_1 \; , \; \ldots \; , \; e_n \;)$$

is a well-formed expression

Tuples (Syntax Rule)

```
<expr> ::= ( <expr> , ... , <expr> )
```

If $e_1, \ldots, e_n$ are well-formed expressions, then

$$(\; e_1 \; , \; \ldots \; , \; e_n \;)$$

is a well-formed expression

Tuple (Typing Rule)

If $e_1, \ldots, e_n$ are of type $\tau_1, \ldots, \tau_n$, respectively, in the context Γ then

$$(\; e_1 \; , \; \ldots \; , \; e_n \;)$$

is of type τ_1 * ... * τ_n in the context Γ

Tuple (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 * \ldots * \tau_n} \text{(tuple)}$$

If $e_1, \ldots, e_n$ are of type $\tau_1, \ldots, \tau_n$, respectively, in the context Γ then

$$(\ e_1 \ , \ \ldots \ , \ e_n \ )$$

is of type τ_1 * ... * τ_n in the context Γ

Tuple (Semantic Rule)

If $e_1, \ldots, e_n$ evaluate to $v_1, \ldots, v_n$, respectively, then

$$(\; e_1 \; , \; \ldots \; , \; e_n \;)$$

evaluates to $(v_1 \; , \; \ldots \; , \; v_n)$

Tuple (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \qquad \ldots \qquad e_n \Downarrow v_n}{(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n)} \ (\mathsf{tupleEval})$$

If $e_1, \ldots, e_n$ evaluate to $v_1, \ldots, v_n$, respectively, then

$$(\ e_1 \ , \ \ldots \ , \ e_n \ )$$

evaluates to $(v_1 \ , \ \ldots \ , \ v_n)$

Example (Typing Derivation)

$$\{x : \mathtt{int}\} \vdash (\ x + x\ ,\ \mathtt{true}\ ) : \mathtt{int} * \mathtt{bool}$$

Example (Typing Derivation)

```
{x : int} ⊢ ( x + x , true ) : int * bool
```

Example (Typing Derivation)

$$\frac{\{\ x : \mathtt{int}\ \} \vdash x + x : \mathtt{int} \qquad \{x : \mathtt{int}\} \vdash \mathtt{true} : \mathtt{bool}}{\{x : \mathtt{int}\} \vdash (\ x + x\ ,\ \mathtt{true}\ ) : \mathtt{int} * \mathtt{bool}}\ (\mathtt{tuple})$$

Example (Typing Derivation)

$$\dfrac{\dfrac{\{x : \mathtt{int}\} \vdash x : \mathtt{int} \qquad \{x : \mathtt{int}\} \vdash x : \mathtt{int}}{\{ x : \mathtt{int} \} \vdash x + x : \mathtt{int}}\,(\mathrm{add}) \qquad \{x : \mathtt{int}\} \vdash \mathtt{true} : \mathtt{bool}}{\{x : \mathtt{int}\} \vdash (\ x + x\ ,\ \mathtt{true}\ ) : \mathtt{int} * \mathtt{bool}}\,(\mathrm{tuple})$$

Example (Typing Derivation)

$$
\dfrac{\dfrac{\dfrac{}{\{x : \mathtt{int}\} \vdash x : \mathtt{int}}\,(\mathtt{var}) \qquad \{x : \mathtt{int}\} \vdash x : \mathtt{int}}{\{ x : \mathtt{int} \} \vdash x + x : \mathtt{int}}\,(\mathtt{add}) \qquad \{x : \mathtt{int}\} \vdash \mathtt{true} : \mathtt{bool}}{\{x : \mathtt{int}\} \vdash (\ x + x\ ,\ \mathtt{true}\ ) : \mathtt{int} \ast \mathtt{bool}}\,(\mathtt{tuple})
$$

Example (Typing Derivation)

$$\dfrac{\dfrac{\dfrac{}{\{x : \mathtt{int}\} \vdash x : \mathtt{int}}\,(\mathtt{var}) \qquad \dfrac{}{\{x : \mathtt{int}\} \vdash x : \mathtt{int}}\,(\mathtt{var})}{\{\ x : \mathtt{int}\ \} \vdash x + x : \mathtt{int}}\,(\mathtt{add}) \qquad \{x : \mathtt{int}\} \vdash \mathtt{true} : \mathtt{bool}}{\{x : \mathtt{int}\} \vdash (\ x + x\ ,\ \mathtt{true}\ ) : \mathtt{int} * \mathtt{bool}}\,(\mathtt{tuple})$$

Example (Typing Derivation)

$$
\dfrac{
  \dfrac{
    \dfrac{}{\{x : \mathtt{int}\} \vdash x : \mathtt{int}}\,(\mathrm{var})
    \quad
    \dfrac{}{\{x : \mathtt{int}\} \vdash x : \mathtt{int}}\,(\mathrm{var})
  }{\{ x : \mathtt{int} \} \vdash x + x : \mathtt{int}}\,(\mathrm{add})
  \quad
  \dfrac{}{\{x : \mathtt{int}\} \vdash \mathtt{true} : \mathtt{bool}}\,(\mathrm{true})
}{\{x : \mathtt{int}\} \vdash (x + x , \mathtt{true}) : \mathtt{int} * \mathtt{bool}}\,(\mathrm{tuple})
$$

Example (Semantic Derivation)

$$\texttt{(2 + 3 , true)} \Downarrow (5, \top)$$

Example (Semantic Derivation)

$$(\ 2\ +\ 3\ ,\ \texttt{true}\ ) \Downarrow (5, \top)$$

Example (Semantic Derivation)

$$\frac{\texttt{2 + 3} \Downarrow 5 \qquad\qquad \texttt{true} \Downarrow \top}{\texttt{(2 + 3 , true)} \Downarrow (5, \top)} \; \texttt{(tupleEval)}$$

Example (Semantic Derivation)

$$\dfrac{\dfrac{\texttt{2} \Downarrow 2 \qquad \texttt{3} \Downarrow 3}{\texttt{2 + 3} \Downarrow 5}\,(\texttt{addEval}) \qquad \texttt{true} \Downarrow \top}{\texttt{(2 + 3 , true)} \Downarrow (5, \top)}\,(\texttt{tupleEval})$$

Example (Semantic Derivation)

$$\dfrac{\dfrac{\dfrac{}{\texttt{2} \Downarrow 2}\,(\texttt{intEval}) \qquad \texttt{3} \Downarrow 3}{\texttt{2 + 3} \Downarrow 5}\,(\texttt{addEval}) \qquad \texttt{true} \Downarrow \top}{\texttt{(2 + 3 , true)} \Downarrow (5, \top)}\,(\texttt{tupleEval})$$

Example (Semantic Derivation)

$$\dfrac{\dfrac{\dfrac{}{\texttt{2} \Downarrow 2}\,(\texttt{intEval}) \qquad \dfrac{}{\texttt{3} \Downarrow 3}\,(\texttt{intEval})}{\texttt{2 + 3} \Downarrow 5}\,(\texttt{addEval}) \qquad \texttt{true} \Downarrow \top}{\texttt{(2 + 3 , true)} \Downarrow (5, \top)}\,(\texttt{tupleEval})$$

Example (Semantic Derivation)

$$\dfrac{\dfrac{\dfrac{}{\texttt{2} \Downarrow 2}\,(\texttt{intEval}) \qquad \dfrac{}{\texttt{3} \Downarrow 3}\,(\texttt{intEval})}{\texttt{2 + 3} \Downarrow 5}\,(\texttt{addEval}) \qquad \dfrac{}{\texttt{true} \Downarrow \top}\,(\texttt{trueEval})}{\texttt{(2 + 3 , true)} \Downarrow (5, \top)}\,(\texttt{tupleEval})$$

Tuple Match (Typing Rule)

If p is of type τ_1 `* … *` τ_n in the context Γ and e is of type τ in the context Γ along with x_i of type τ_i, then

`match` p `with |` x_1 `, … ,` x_n `->` e

is of type τ in the context Γ

Tuple Match (Typing Rule)

$$\frac{\Gamma \vdash p : \tau_1 * \ldots * \tau_n \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \textsf{match}\ p\ \textsf{with}\ |\ x_1, \ldots, x_n \to e : \tau}\ (\textsf{matchTuple})$$

If p is of type τ_1 * … * τ_n in the context Γ and e is of type τ in the context Γ along with x_i of type τ_i, then

match p with | x_1 , … , x_n -> e

is of type τ in the context Γ

Tuple Match (Semantic Rule)

If p evaluates to $(v_1, \ldots, v_n)$ and e evaluates to v after substituting v_i for x_i ($\forall i$), then

`match` p `with |` x_1 `,…,` x_n `->` e

evaluates to v

Tuple Match (Semantic Rule)

$$\frac{p \Downarrow (v_1, \ldots, v_n) \qquad e' = [v_1/x_1] \ldots [v_n/x_n]e \qquad e' \Downarrow v}{(\textsf{match}\ p\ \textsf{with} \mid x_1, \ldots, x_n \to e) \Downarrow v}\ (\textsf{matchTupleEval})$$

If p evaluates to $(v_1, \ldots, v_n)$ and e evaluates to v after substituting v_i for x_i ($\forall i$), then

`match` p `with |` x_1 `,…,` x_n `->` e

evaluates to v

Practice Problem

$\emptyset \vdash$ `fun x -> match x with | (a, b) -> a + b :` τ

Determine the type τ so that the above judgment is derivable from the rules below. Also give a derivation

$$\frac{\Gamma \vdash p : \tau_1 * \ldots * \tau_n \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \text{match } p \text{ with } \mid x_1, \ldots, x_n \rightarrow e : \tau} \text{(matchTuple)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{(addInt)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2} \text{(fun)}$$

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \text{(var)}$$

$\varnothing \vdash$ `fun x -> match x with | (a, b) -> a + b` $: \tau$

Solution

Homework

Implement the function

```
val fill_in_steps : (int, int, int) -> int list
```

so that `fill_in_steps (a, b, c)` *is the shortest list of numbers starting at* `a`*, ending at* `c`*, containing* `b` *and having consecutive adjacent elements*

example: `fill_in_steps (1, 4, -2) = [1;2;3;4;3;2;1;0;-1;-2]`

Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

Records are ***unordered*** ***labeled*** fixed-length heterogeneous collections of data

They are useful for organizing large collections of data (akin to database records)

Record Syntax (Informal)

```
type record_ty =
  {
    field1 : ty1;
    field2 : ty2;
    ...
    fieldn : tyn;
  }
```

```
let record_expr : record_ty =
  {
    field1 = expr1;
    field2 = expr2;
    ...
    fieldn = exprn;
  }
```

For a record, we have to specify the type of each field

When we construct a record, every field must have a value

Accessors

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

Records support **dot-notation**

(we can also access records by pattern matching)

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"***u*** *with* ***num_posts*** *incremented, keep everything else the same*"

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"***u*** *with* ***num_posts*** *incremented, keep everything else the same*"

Data in functional languages are immutable. This returns a new record with the update

demo

(tuples and records)

Homework

» Define formal syntax rules for records

» Define formal typing rules for records

» Define formal semantics rules for records

Unions

Second Greatest Feature of OCaml

Simple Variants

```
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

First letter of type names is **lower_case** and Constructor names is **Upper_case**

Simple Variants

constructor

```
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

First letter of type names is **lower_case** and Constructor names is **Upper_case**

Pattern Matching

```
let supported (sys : os) : bool =
  match sys with
  | BSD -> false
  | _ -> true
```

We work with variants by **pattern matching:**

» giving a pattern that a value can match with

» writing what to do for each pattern

Pattern Matching

```
                  let supported (sys : os) : bool =
                    match sys with
constant pattern  | BSD -> false
wildcard pattern  | _ -> true
```

We work with variants by **pattern matching:**

» giving a pattern that a value can match with

» writing what to do for each pattern

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Variants can carry data, which allows us to represent more complex structures

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Note the syntax

Variants can carry data, which allows us to represent more complex structures

Practice Problem

```
let area (s : shape) =
  match s with
  | Rect r -> r.base *. r.height
  | Triangle { sides = (a, b) ; angle } -> Float.sin angle *. a *. b
  | Circle r -> r *. r *. Float.pi
```

Define the variant ***shape*** *which makes this function type-check*

What about variable length data?

Recursive ADTs

Greatest Feature of OCaml

Example: Lists

```
type intlist
  = Nil
  | Cons of int * intlist

let example = Cons (1, Cons (2, Cons (3, Nil)))
```

The type **intlist** is available as the type of data which a constructor of **intlist** holds

We can use recursive ADTs to create variable-length data types

Example: Lists

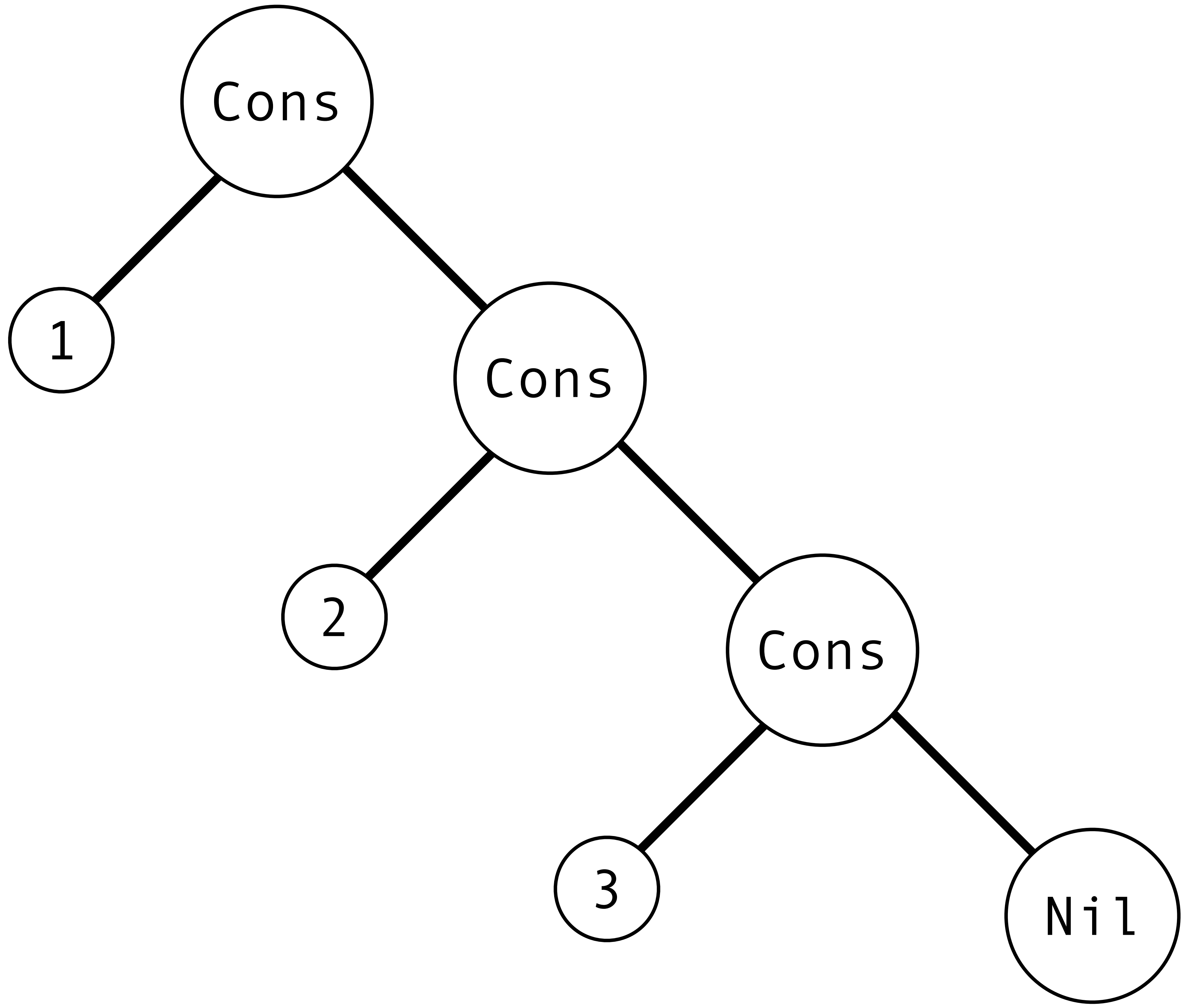
```
type intlist
  = Nil
  | Cons of int * intlist

let example = Cons (1, Cons (2, Cons (3, Nil)))
```

The type **intlist** is available as the type of data which a constructor of **intlist** holds

We can use recursive ADTs to create variable-length data types

The Picture

```
Cons (1,
  Cons (2,
    Cons (3,
      Nil)))
```



We think of values of recursive variants as trees with constructors as nodes and carried data as leaves

demo

(snoc for intlist)

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

Suppose we're building a calculator*

*This is exactly what we'll be doing when we build an interpreter.

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

Suppose we're building a calculator*

Before we compute the value of an input, we first have to find an abstract representation of the input

*This is exactly what we'll be doing when we build an interpreter.

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

Suppose we're building a calculator*

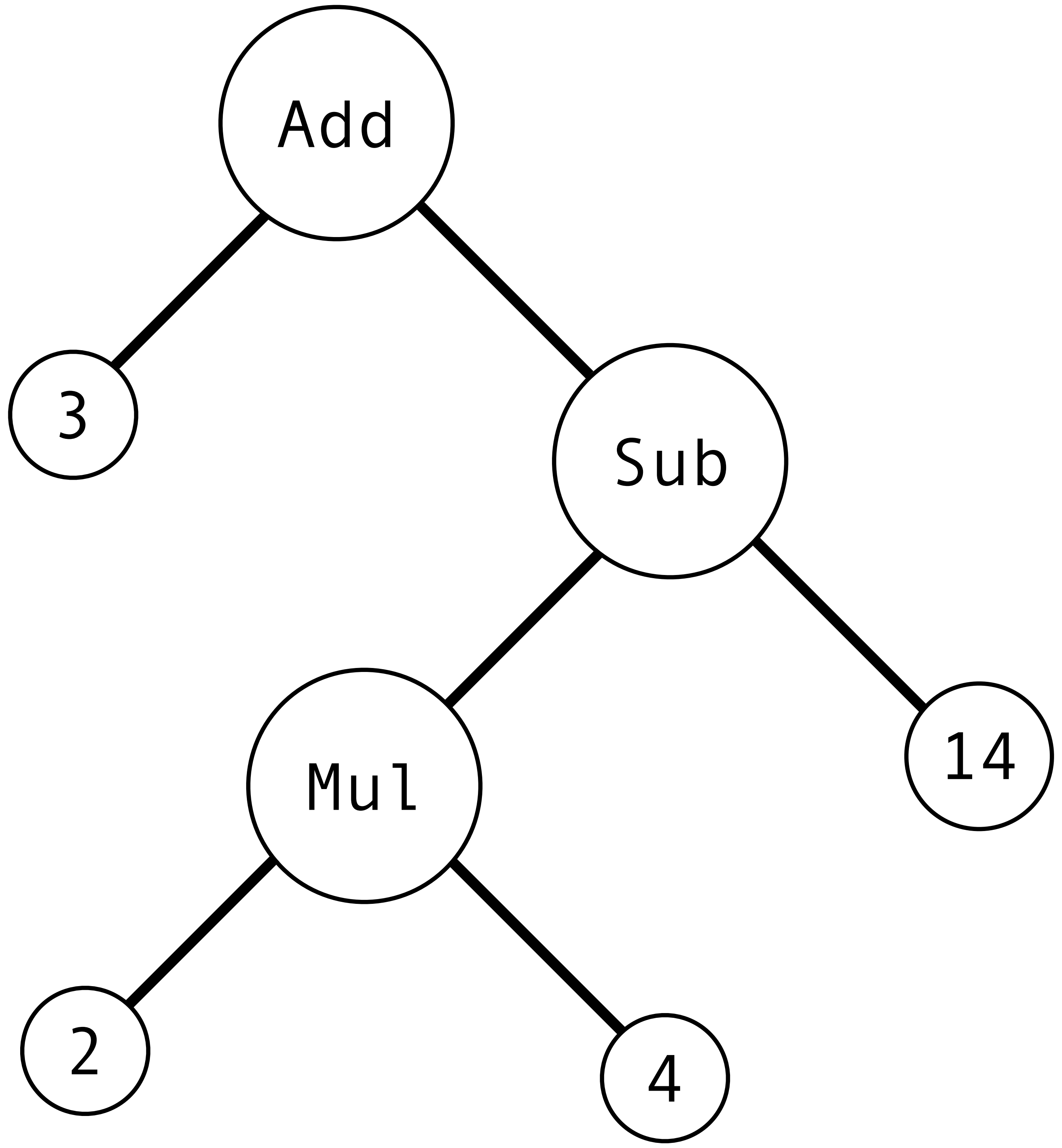
Before we compute the value of an input, we first have to find an abstract representation of the input

This will help us separate the tasks of evaluation and parsing

*This is exactly what we'll be doing when we build an interpreter.

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14) \Rightarrow$$



We can represent an expression abstractly as a tree with operations as nodes and number values as leaves

A More Interesting Example: Expressions

```
type expr
  = Val of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr

let _ = Add (Val 3, Sub (Mul (Val 2, Val 4), Val 14))
```

Which means we can represent it as a recursive variant!

Parametrized ADTs

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Parameterized Variants

type variable

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Parameterized Variants

type variable

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist
```

type constructor

```
let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Parametric Polymorphism

```
let rev_tail (l : 'a list) : 'a list =
  let rec go acc l =
    match l with
    | [] -> acc
    | x :: xs -> go (x :: acc) xs
  in go [] l
```

Parametric Polymorphism

```
let rev_tail (l : 'a list) : 'a list =
  let rec go acc l =
    match l with
    | [] -> acc
    | x :: xs -> go (x :: acc) xs
  in go [] l
```

This allows us to write functions which can be more generally applied (reversing a list **does not depend on** what's in the list)

Parametric Polymorphism

```
let rev_tail (l : 'a list) : 'a list =
  let rec go acc l =
    match l with
    | [] -> acc
    | x :: xs -> go (x :: acc) xs
  in go [] l
```

This allows us to write functions which can be more generally applied (reversing a list **does not depend on** what's in the list)

Note. Because of **type-inference**, we rarely have to think about this

We'll come back to this next time...

Summary

Tuples, records, and ADTs help us organize data and create abstract interfaces

Recursive and parametrized ADTs give us richer structure