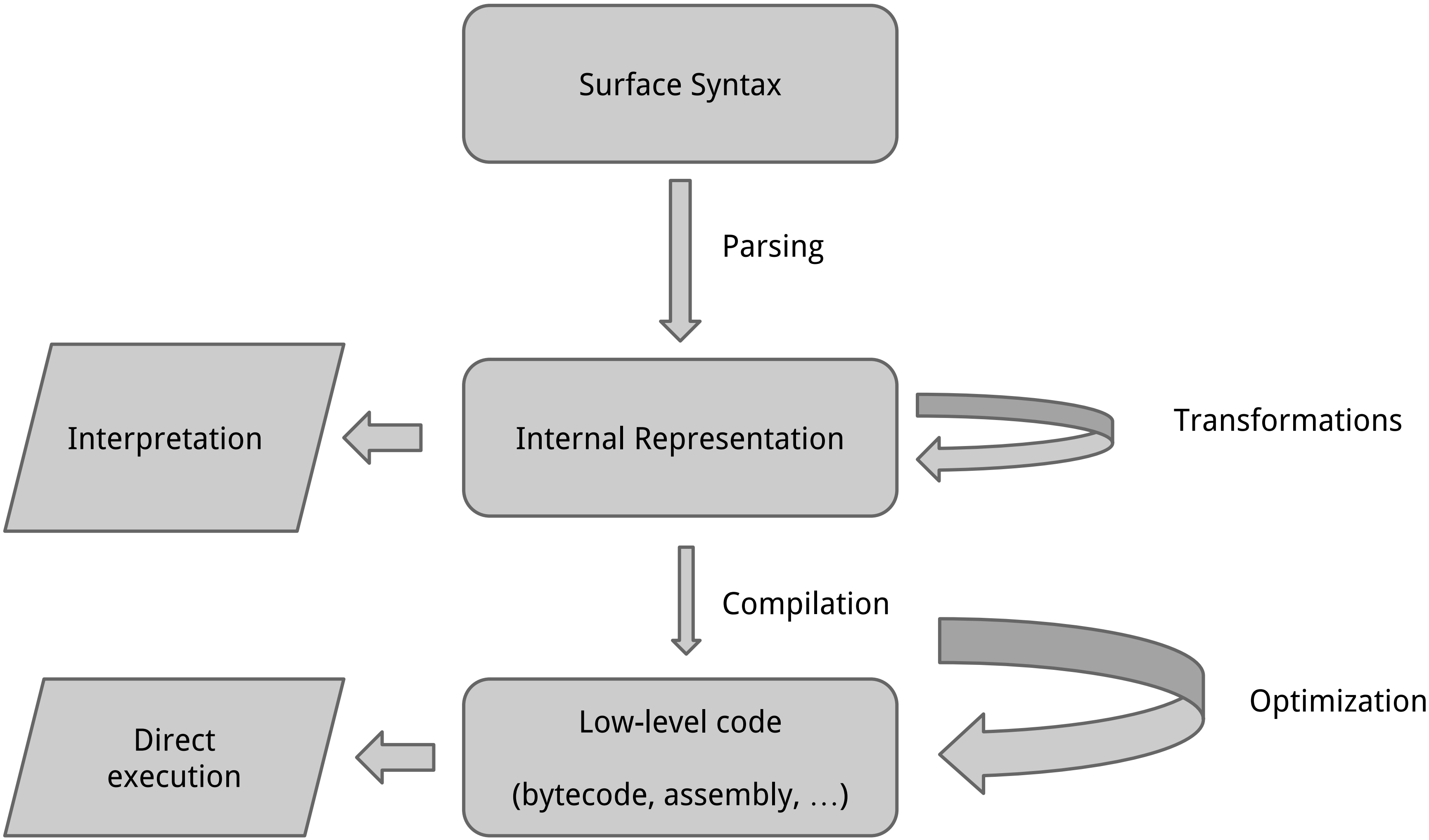
Introduction to Interpretation

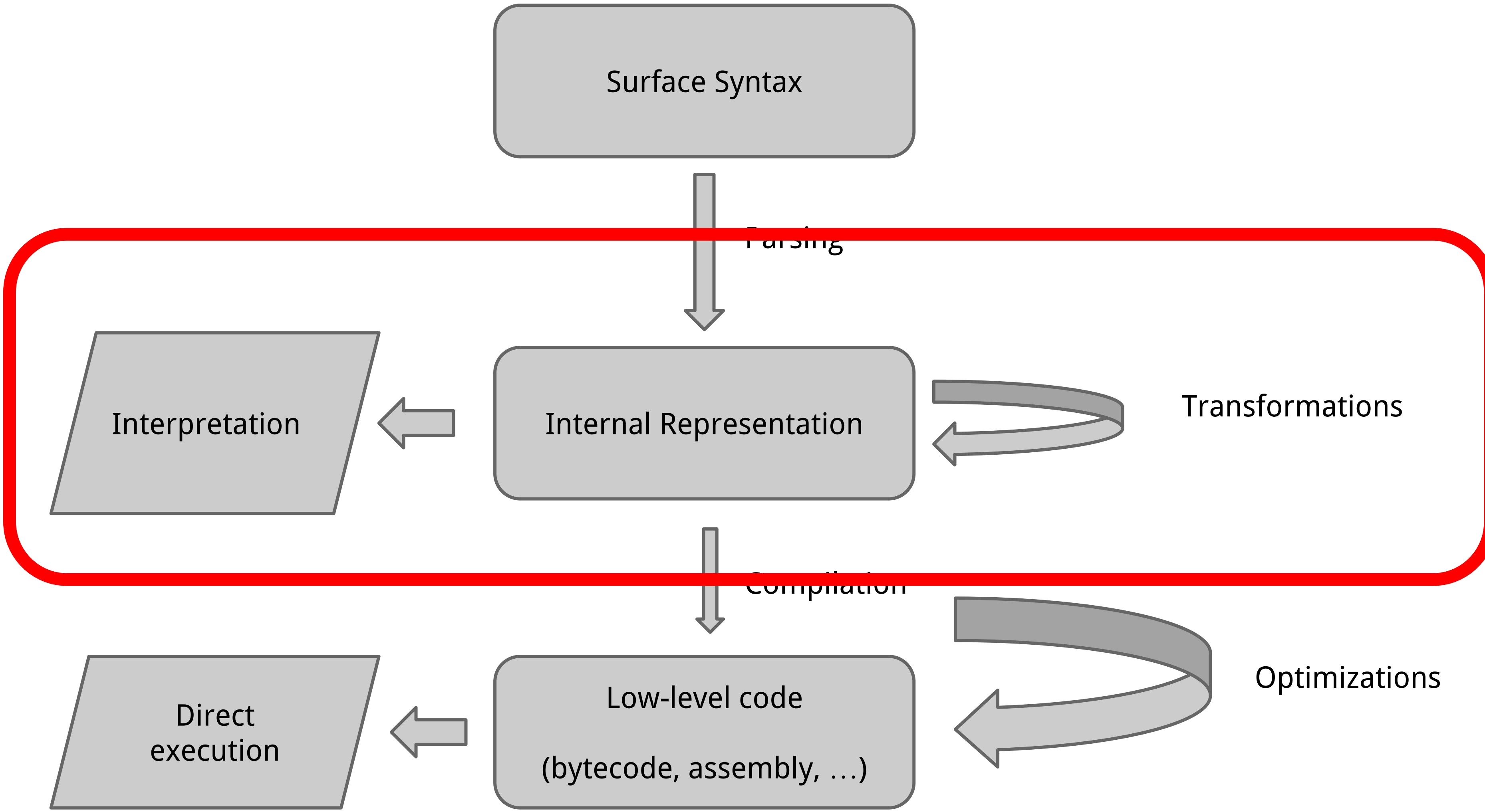
January 24, 2014

Riccardo Pucella

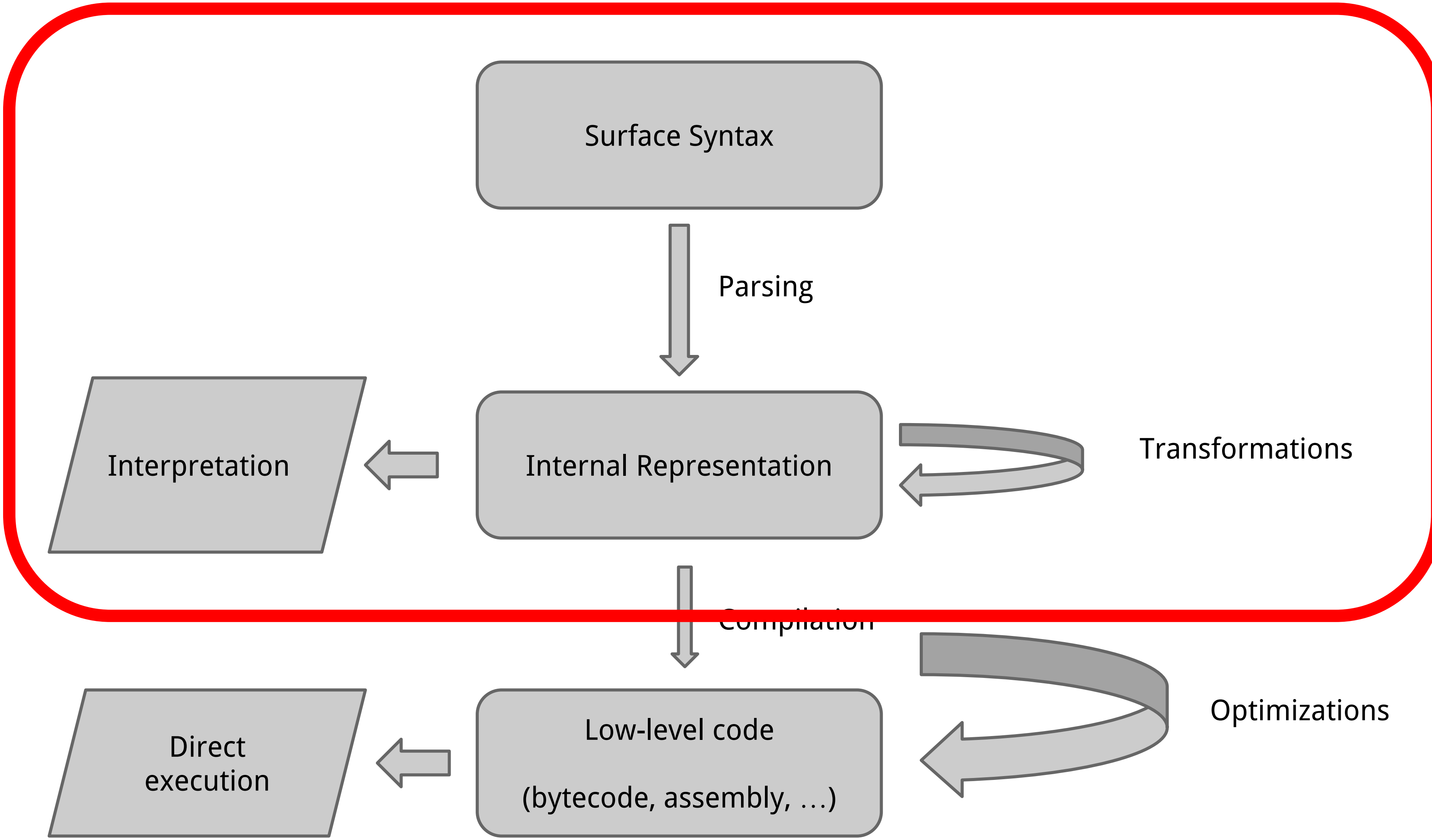
The structure of language execution



The structure of language execution



The structure of language execution



Standard ML

- Our implementation language
- A mostly-functional statically-typed higher-order language
- Shell for interactive use
- Static type system with
  - Type inference
  - Polymorphic types

Basic types, as usual

Integers, reals, Booleans, strings, …

```
1 : int
2.3 : real
true : bool
“hello” : string
```

Tuples

```
(1,2,3) : int * int * int
(1,”hello”) : int * string
```

Programs

A program is an expression that evaluates to a value:

```
1 + (3 * 4 / 2)
size “hello”
size “hello” + 4
```

Conditionals, local declarations:

```
if x then 3 else (4 * b)
let val x = 10 in x * x end
```

Top-level declarations

At the interactive shell, can define names for values—these are immutable:

```
val x = “hello”
val y = size x
```

Can also define functions:

```
fun ctof c = (1.8 * c) + 32.0
fun attach s1 s2 = s1^”,”^s2
```

Pattern matching

Can define functions with pattern matching:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-2) + fib (n-1)
```

```
fun sum 0 _ _ = 0
  | sum _ 0 _ = 0
  | sum _ _ 0 = 0
  | sum x y z = x + y + z
```

Pattern matching

Can also have pattern matching elsewhere than function declarations:

```
case t of (0,_,_) => 0
        | (_,0,_) => 0
        | (_,_,0) => 0
        | (x,y,z) => x+y+z
```

That's usually how you access tuple elements

Lists

A (linked) list is a sequence of elements of the same type.

```
[1, 2, 3]
[“hello”, “world”]
[]
```

Infix operator `::` to create a list from an element and a list:

```
1 :: []
2 :: x
```

Functions on lists

Use pattern matching to analyze the list(s):

```
fun length [] = 0
  | length (x::xs) = 1 + (length xs)

fun append [] ys = ys
   | append (x::xs) ys = x::(append xs ys)

fun reverse [] = []
   | reverse (x::xs) = append (reverse xs) [x]
```

These functions have polymorphic types

Exercises

Write functions

```
scale : int -> int list -> int list
   scale 10 [1, 2, 3] = [10, 20, 30]

add : int list -> int list -> int list
  add [1,2,3] [4,5,6] = [5,7,9]

inner : int list -> int list -> int
  inner [1,2,3] [4,5,6] = 4 + 10 + 18 = 32
```

Datatypes

Define variant types (union types):

```
datatype ‘a tree = Empty
                 | Node of ‘a * ‘a tree * ‘a tree

Empty
Node (1, Node (2, Empty, Empty),
         Node (3, Empty, Empty))
Node (1,Node(2,Node(3,Empty),Empty),Empty)
```

Lists are a built-in form of these

Pattern-matching for datatypes

```
fun height Empty = 0
  | height (Node (v,left,right)) =
      1 + Int.max (height left, height right)


fun find x Empty = false
  | find x (Node (v,left,right)) =
      (x=v) orelse
       (find x left) orelse
        (find x right)
```

A simple mathematical language

- We’re going to build-up a small language of mathematical expressions

- Computing over integers (extended later)
  - Operations +, -, *

- Internal representation needs to account for the fact that expressions can be nested
  - (3 + 4) * (5 + 6)

Internal representation

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
```

Interpretation via an evaluation function:

```
eval : expr -> int
```

Evaluation function

```
fun eval (EInt i) = i
  | eval (EAdd (e,f)) = eval e + eval f
  | eval (ESub (e,f)) = eval e - eval f
  | eval (EMul (e,f)) = eval e * eval f
  | eval (ENeg e) = ~ (eval e)
```

Evaluation function

```
fun applyAdd i j = i + j
fun applySub i j = i - j
fun applyMul i j = i * j
fun applyNeg i = ~ i

fun eval (EInt i) = i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
```

Vectors

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
```

```
val eval : expr -> int
```

Vectors

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EVec of int list
```

```
eval : expr -> ??
```

Vectors

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EVec of int list

datatype value = VInt of int
               | VVec of int list

   eval : expr -> value
```

Evaluation function

```
fun eval (EInt i) = i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
  | eval (EVec v) = VVec v
```

Evaluation function helpers

```
fun applyAdd i j = i + j
```

Evaluation function helpers

```
exception TypeError of string


fun applyAdd (VInt i) (VInt j) = VInt (i + j)
  | applyAdd (VVec v) (VVec w) =
       if length v = length w
       then VVec (addVec v w)
       else raise TypeError "applyAdd”
  | applyAdd _ _ = raise TypeError "applyAdd”
```

Evaluation function helpers

```
fun applyMul i j = i * j
```

Evaluation function helpers

```
fun applyMul (VInt i) (VInt j) = VInt (i * j)
  | applyMul (VInt i) (VVec v) = VVec (scaleVec i v)
  | applyMul (VVec v) (VInt i) = VVec (scaleVec i v)
  | applyMul (VVec v) (VVec w) =
      if length v = length w
      then VInt (inner v w)
      else raise TypeError "applyMul"
  | applyMul _ _ = raise TypeError "applyMul"
```

Evaluation function helpers

```
fun applyNeg i = ~ i
```

```
fun applySub i j = i - j
```

Evaluation function helpers

```
fun applyNeg (VInt i) = VInt (~ i)
  | applyNeg (VVec v) = VVec (scaleVec ~1 v)
  | applyNeg _ = raise TypeError "applyNeg"


fun applySub i j = applyAdd i (applyNeg j)
```

Booleans and conditionals

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EVec of int list



datatype value = VInt of int
               | VVec of int list
```

Booleans and conditionals

```
datatype expr = EInt of int
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EVec of int list
              | EBool of bool
              | EIf of expr * expr * expr

datatype value = VInt of int
               | VVec of int list
               | VBool of bool
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
  | eval (EVec v) = VVec v
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
  | eval (EVec v) = VVec v
  | eval (EBool b) = VBool b
  | eval (EIf (e,f,g)) = evalIf (eval e) f g
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ENeg e) = ApplyNeg (eval e)
  | eval (EVec v) = VVec v
  | eval (EBool b) = VBool b
  | eval (EIf (e,f,g)) = evalIf (eval e) f g

and evalIf (VBool true) f g = eval f
  | evalIf (VBool false) f g = eval g
  | evalIf _ _ _ = raise TypeError "evalIf"
```

First homework

- Some exercises on SML

- Extend the expression language with
  - Operations on Booleans
  - Matrix operations
  - Division and rational numbers