Learn Haskell

“Code you can believe in”

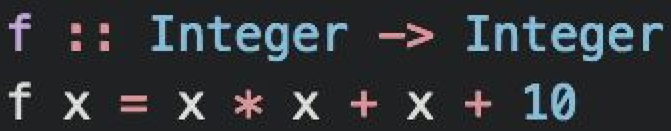
Leo Zhang

What is Haskell

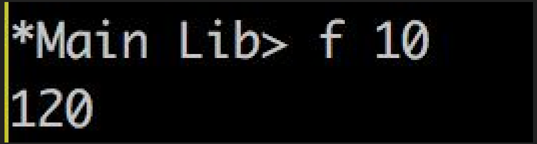
Statically typed pure functional language

Function definition

Haskell

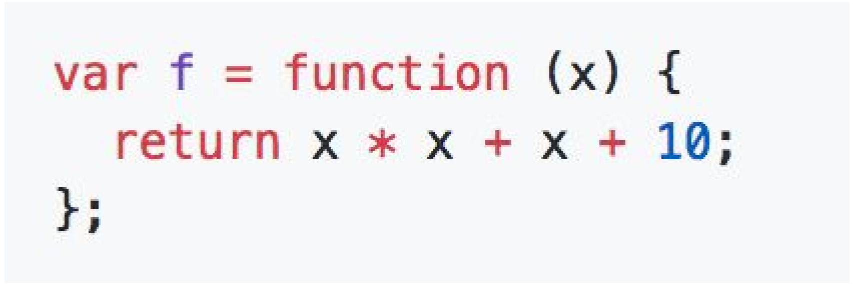


```haskell
f :: Integer -> Integer
f x = x * x + x + 10
```

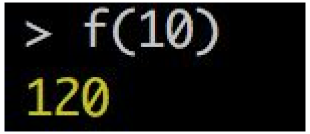


```
*Main Lib> f 10
120
```

Javascript



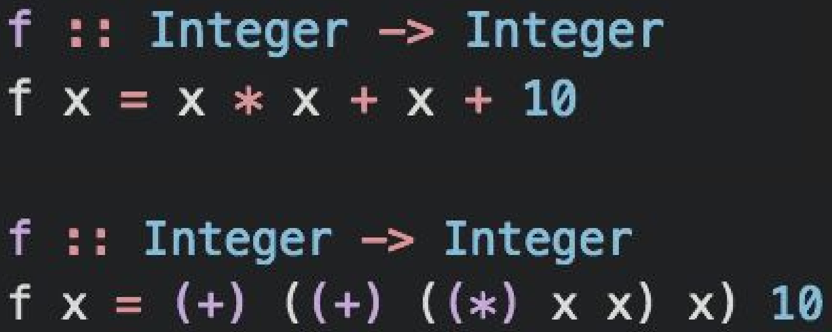
```javascript
var f = function (x) {
  return x * x + x + 10;
};
```



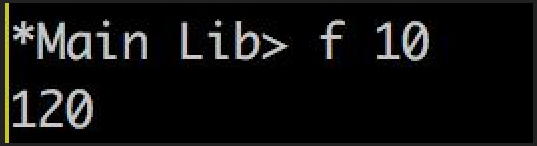
```
> f(10)
120
```

Operators are also functions

Haskell

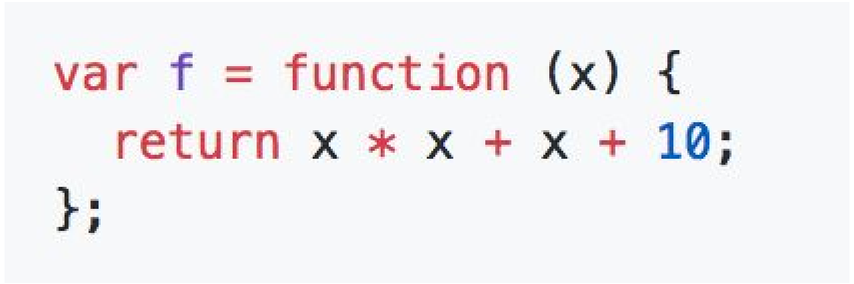


```haskell
f :: Integer -> Integer
f x = x * x + x + 10

f :: Integer -> Integer
f x = (+) ((+) ((*) x x) x) 10
```

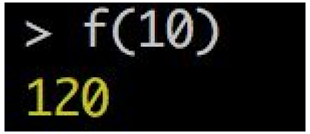


```
*Main Lib> f 10
120
```

Javascript



```javascript
var f = function (x) {
  return x * x + x + 10;
};
```

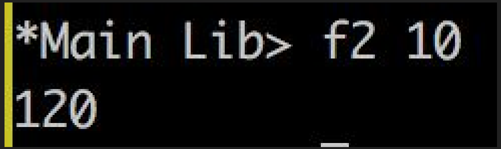


```
> f(10)
120
```

Define variables

Haskell

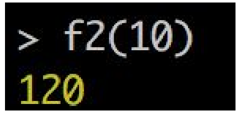
```haskell
f2 :: Integer -> Integer
f2 x = x2 + x + 10
  where
    x2 = sqaure x
    sqaure n = n * n
```



```
*Main Lib> f2 10
120
```

Javascript

```javascript
var f2 = function (x) {
  var square = function(n) {
    return n * n;
  };

  var x2 = square(x);
  return x2 + x + 10;
};
```

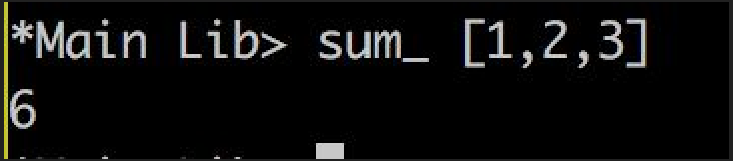


```
> f2(10)
120
```

Recursion is the building block

Haskell

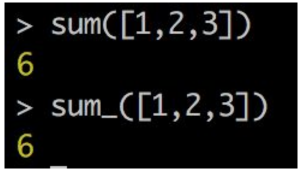
```haskell
sum_ :: [Integer] -> Integer
sum_ xs = recur 0 xs
  where
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```



```
*Main Lib> sum_ [1,2,3]
6
```

Javascript

```javascript
// iteration with for-loop
var sum = function (arr) {
  var total= 0;
  for (var i = 0; i < arr.length; i++) {
    total += arr[i];
  }
  return total;
};

// recursion
var sum_ = function(arr) {
  var recur = function(total, arr) {
    if (arr.length === 0) {
      return total;
    }
    var next = arr[0];
    return recur(total + next, arr.slice(1));
  };
  return recur(0, arr);
};
```

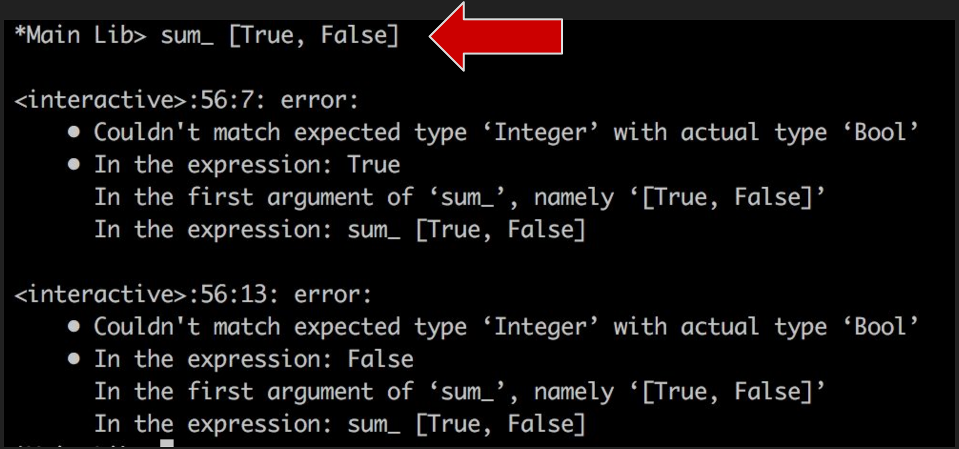


```
> sum([1,2,3])
6
> sum_([1,2,3])
6
```

Type Check

Haskell

```haskell
sum_ :: [Integer] -> Integer
sum_ xs = recur 0 xs
  where
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```



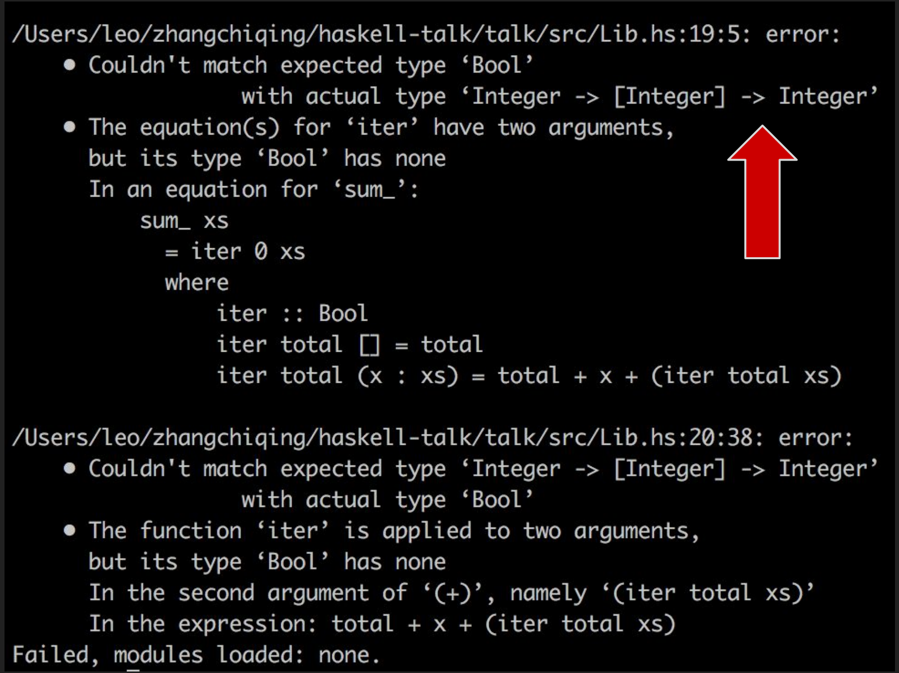
```
*Main Lib> sum_ [True, False]

<interactive>:56:7: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘Bool’
    • In the expression: True
      In the first argument of ‘sum_’, namely ‘[True, False]’
      In the expression: sum_ [True, False]

<interactive>:56:13: error:
    • Couldn't match expected type ‘Integer’ with actual type ‘Bool’
    • In the expression: False
      In the first argument of ‘sum_’, namely ‘[True, False]’
      In the expression: sum_ [True, False]
```

Type Inference

```haskell
sum_ :: [Integer] -> Integer
sum_ xs = recur 0 xs
  where
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```

```haskell
sum_ xs = recur 0 xs
  where
    recur :: Bool
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```

```haskell
sum_ xs = recur 0 xs
  where
    recur :: Integer -> [Integer] -> Integer
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```

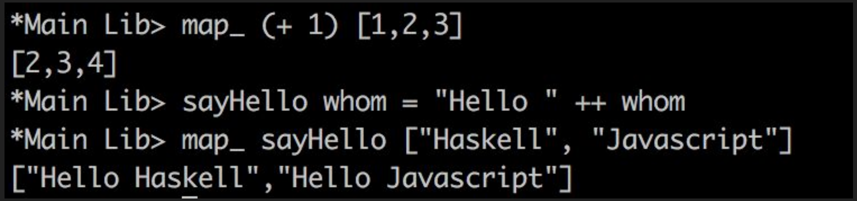


```
/Users/leo/zhangchiqing/haskell-talk/talk/src/Lib.hs:19:5: error:
    • Couldn't match expected type ‘Bool’
                  with actual type ‘Integer -> [Integer] -> Integer’
    • The equation(s) for ‘iter’ have two arguments,
      but its type ‘Bool’ has none
      In an equation for ‘sum_’:
          sum_ xs
            = iter 0 xs
            where
                iter :: Bool
                iter total [] = total
                iter total (x : xs) = total + x + (iter total xs)

/Users/leo/zhangchiqing/haskell-talk/talk/src/Lib.hs:20:38: error:
    • Couldn't match expected type ‘Integer -> [Integer] -> Integer’
                  with actual type ‘Bool’
    • The function ‘iter’ is applied to two arguments,
      but its type ‘Bool’ has none
      In the second argument of ‘(+)’, namely ‘(iter total xs)’
      In the expression: total + x + (iter total xs)
Failed, modules loaded: none.
```

Type Inference

Haskell

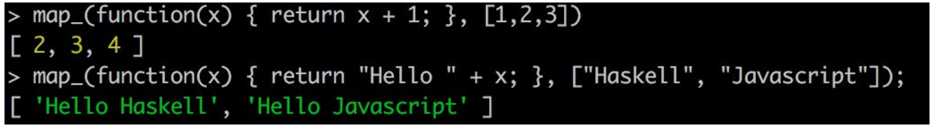
```haskell
map_ :: (a -> b) -> [a] -> [b]
map_ f [] = []
map_ f (x:xs) = f x : map_ f xs
```



```
*Main Lib> map_ (+ 1) [1,2,3]
[2,3,4]
*Main Lib> sayHello whom = "Hello " ++ whom
*Main Lib> map_ sayHello ["Haskell", "Javascript"]
["Hello Haskell","Hello Javascript"]
```

Javascript

```javascript
var map_ = function(f, arr) {
  var recur = function(f, mapped, remaining) {
    if (remaining.length === 0) {
      return mapped;
    }
    var next = f(remaining[0]);
    return recur(f, mapped.concat(next), remaining.slice(1));
  };
  return recur(f, [], arr);
};
```

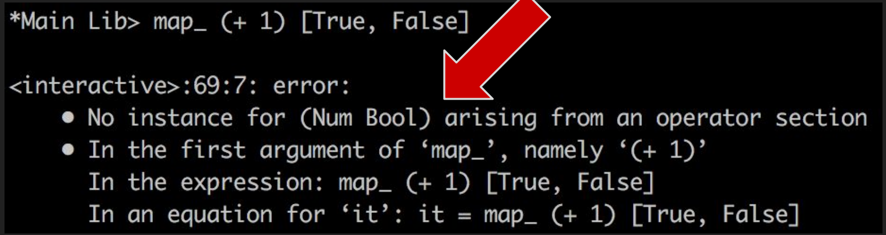


```
> map_(function(x) { return x + 1; }, [1,2,3])
[ 2, 3, 4 ]
> map_(function(x) { return "Hello " + x; }, ["Haskell", "Javascript"]);
[ 'Hello Haskell', 'Hello Javascript' ]
```

Type Inference

Haskell

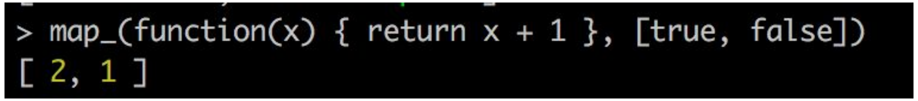
```haskell
map_ :: (a -> b) -> [a] -> [b]
map_ f [] = []
map_ f (x:xs) = f x : map_ f xs
```



```
*Main Lib> map_ (+ 1) [True, False]

<interactive>:69:7: error:
    • No instance for (Num Bool) arising from an operator section
    • In the first argument of ‘map_’, namely ‘(+ 1)’
      In the expression: map_ (+ 1) [True, False]
      In an equation for ‘it’: it = map_ (+ 1) [True, False]
```

Javascript

```javascript
var map_ = function(f, arr) {
  var recur = function(f, mapped, remaining) {
    if (remaining.length === 0) {
      return mapped;
    }
    var next = f(remaining[0]);
    return recur(f, mapped.concat(next), remaining.slice(1));
  };
  return recur(f, [], arr);
};
```



```
> map_(function(x) { return x + 1 }, [true, false])
[ 2, 1 ]
```

Compose High Order Functions

Haskell

Javascript

```haskell
map_ :: (a -> b) -> [a] -> [b]
map_ f [] = []
map_ f (x:xs) = f x : map_ f xs
```

```haskell
sum_ :: [Integer] -> Integer
sum_ xs = recur 0 xs
  where
    recur total [] = total
    recur total (x:xs) = total + x + (recur total xs)
```

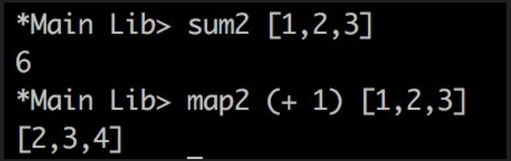
```javascript
var map_ = function(f, arr) {
  var recur = function(f, mapped, remaining) {
    if (remaining.length === 0) {
      return mapped;
    }
    var next = f(remaining[0]);
    return recur(f, mapped.concat(next), remaining.slice(1));
  };
  return recur(f, [], arr);
};
```

```javascript
// recursion
var sum_ = function(arr) {
  var recur = function(total, arr) {
    if (arr.length === 0) {
      return total;
    }
    var next = arr[0];
    return recur(total + next, arr.slice(1));
  };
  return recur(0, arr);
};
```

Compose High Order Functions

Haskell

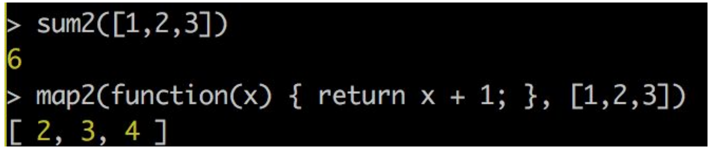
```haskell
fold_ :: (a -> b -> b) -> b -> [a] -> b
fold_ f accum [] = accum
fold_ f accum (x:xs) = f x (fold_ f accum xs)

sum2 :: [Integer] -> Integer
sum2 xs = fold_ (+) 0 xs

map2 :: (a -> b) -> [a] -> [b]
map2 f xs = fold_ acc [] xs
  where
    acc x accum = f x : accum
```



```
*Main Lib> sum2 [1,2,3]
6
*Main Lib> map2 (+ 1) [1,2,3]
[2,3,4]
```

Javascript

```javascript
var fold_ = function(f, accum, arr) {
  if (arr.length === 0) {
    return accum;
  }
  var next = arr[0];
  return f(next, fold_(f, accum, arr.slice(1)));
};

var sum2 = function(arr) {
  return fold_(function(n, accum) {
    return n + accum;
  }, 0, arr);
};

var map2 = function(f, arr) {
  return fold_(function(n, accum) {
    accum.splice(0, 0, f(n));
    return accum;
  }, [], arr);
};
```



```
> sum2([1,2,3])
6
> map2(function(x) { return x + 1; }, [1,2,3])
[ 2, 3, 4 ]
```

Key takeaways

- Recursion is the fundamental building block
- Compose high order functions
- Static typed and advanced type inference system

“Code you can believe in”

Resources

- Real world Haskell
  http://book.realworldhaskell.org/read/
- Learn you a Haskell for Great Good!
  http://learnyouahaskell.com/chapters
- What I Wish I Knew When Learning Haskell
  http://dev.stephendiehl.com/hask/