Example Scheme programs

```
;;; Using built-in integers
(define (factorial n)
  (if (eq? n 0)
        1
        (* n (factorial (- n 1)))))
```

```
;;; cons is the only provided way build data structures
(define (zip l0 l1)"
     (if (null? l0) "
        (list)"
         (if (null? l1)"
             (list)"
             (cons (cons (car l0) (car l1)) (zip (cdr l0) (cdr l1)))))"
```

empty list val

(~ nil, '())

A Haskell data type for Scheme expressions

```haskell
data Exp
  = Atom String
  | List [Exp]
  | Number Integer
  | String String
  | Nil
  | Bool Bool
  deriving (Eq, Ord, Show)
```

(cons 1 "foo")

rep'd List [Atom "cons", Number 1, String "foo"]

∈ Exp

(Bool True) reps #t

Parse of "(define (factorial ..."

(define ...) [2, 3, 0]

```
List
    [ Atom "define"
    , List [ Atom "factorial" , Atom "n" ]
    , List
        [ Atom "if"
        , List [ Atom "eq?" , Atom "n" , Number 0 ]
        , Number 1
        , List
            [ Atom "*"
            , Atom "n"
            , List
                [ Atom "factorial"
                , List [ Atom "-" , Atom "n" , Number 1
                    ]
                ]
            ]
        ]
    ]
```

(factorial n)

index 2 →

3 →

0

"addressed" subexp

Tree addressing

The result returned by `parseExp :: String -> Exp` is called a *parse tree* or an **abstract syntax tree**, or **AST**.

It will be convenient to refer to subtrees using tree *paths*.
Convenient fact: Scheme AST nodes are simply either leaves or a list of subtrees.

```
type Path = [Int]
```

To follow a path `ks = [k1, k2, ...]` in an expression `e`:

- if `ks = []` then the addressed subexpression is `e` itself
- otherwise, `e` must be a list; follow [k2,...] starting at the k1-th element.

Exercise: locate the expression at the path address [2,3,0] on the previous slide.

It's convenient to package together an expression and a path into it.

```
data Lens = Lens
  { lensExp :: Exp
  , lensPath :: Path
  }
```

~ data Lens = Lens Exp Path

e.g. lens = { lensExp = Number 0, lensPath = [3] }

= Lens (Number 0) [3]

Two fundamental operations on lenses:

*

```
-- Get the subexpression addressed by the path.
get :: Lens -> Exp

-- Replace (can't really "set" anything in Haskell) the addressed subexpression
set :: Lens -> Exp -> Exp
```

could have used lens here

```
ghci> lens0 = fromJust $ findSubexp (== Atom "*") fact
Lens {lensExp = List [Atom "define", ...], lensPath = [2,3,0]}
```

factorial

The rest of the lecture is about the interpreter in Assignment 9.

See the attached Haskell file(s).