Goals

- Define-datatype
- Concrete Syntax vs. Abstract Syntax
- Practice with recursive programs

A-exp = number
  | A-exp + A-exp
  | A-exp - A-exp
  | A-exp * A-exp
  | A-exp / A-exp
  | (A-exp)

Concrete syntax of arithmetic expressions
(1 + 2) * 4 - 64 / 2

A-exp = number
  | +(A-exp, A-exp)
  | -(A-exp, A-exp)
  | *(A-exp, A-exp)
  | /(A-exp, A-exp)

Alternate concrete syntax for arithmetic expressions
-(*(+(1, 2), 4),
  /(64, 2))

Concrete syntax of a language specifies exactly how to write down an expression of that language.

To write a program that operates on a language we need to represent expressions in that language as computer data.

Such a representation is called an abstract syntax.

Since programming languages are usually tree-like, we call the internal representation of an expression an Abstract Syntax Tree (AST)


```
A-expAST = number
         | (list '+ A-expAST A-expAST)
         | (list '- A-expAST A-expAST)
         | (list '* A-expAST A-expAST)
         | (list '/ A-expAST A-expAST)
```

An abstract syntax for A-exp.

```
(list '- (list '* (list '+ 1 2) 4)
         (list '/ 64 2))
```

The correspondence between concrete and abstract syntax is not always so obvious.

ASIDE:

A compiler course usually spends a considerable time on parsing, the translation from concrete to abstract syntax.

Thursday you will see a tool to parse a language like our second A-exp example.

Let's do something simple.
Write a program to count the number of + operators in an A-exp

Our abstract syntax is lacking:

- Lots of car cdr cadar caddr caddar, etc...
- No nmemonic clues as to what each part of the tree represents.
- Not much error checking

Define-datatype to the rescue.

```
(define-datatype a-exp a-exp?
  [num (val number?)]
  [plus (lhs a-exp?)
        (rhs a-exp?)]
  [minus (lhs a-exp?)
         (rhs a-exp?)]
  [times (lhs a-exp?)
         (rhs a-exp?)]
  [divide (lhs a-exp?)
          (rhs a-exp?)])

(minus (times (plus (num 1) (num 2))
              (num 4))
       (divide (num 64) (num 2)))
```


(num 'a) ⇒ error
(plus 3 4) ⇒ error
...

Define-datatype has a counterpart: cases

```
(define (count+ exp)
  (cases a-exp exp
    [num (n) 0]
    [plus (l r) (+ 1 (count+ l) (count+ r))]
    [minus (l r) (+ (count+ l) (count+ r))]
    [times (l r) (+ (count+ l) (count+ r))]
    [divide (l r) (+ (count+ l) (count+ r))]
    [else (error 'count+ "given an unknown a-exp")]))
```

General form of define-datatype and cases

Define-datatype provides a mechanism for defining and building trees (including ASTs)
Cases provides a mechanism for extracting information from a define-datatype tree

Let's build an evaluator for arithmatic expressions

This is how we might write a BNF for a-exp as defined above (with 2 extensions)

```
va-exp = (num number)
       | (plus va-exp va-exp)
       | (minus va-exp va-exp)
       | (times va-exp va-exp)
       | (divide va-exp va-exp)
       | (pow va-exp va-exp)
       | (var symbol)
```

Here is the extended define-dataype:

```
(define-datatype a-exp a-exp?
  [num (val number?)]
  [plus (lhs a-exp?)
        (rhs a-exp?)]
  [minus (lhs a-exp?)
         (rhs a-exp?)]
  [times (lhs a-exp?)
         (rhs a-exp?)]
  [divide (lhs a-exp?)
          (rhs a-exp?)]
  [pow (lhs a-exp?)
       (rhs a-exp?)]
  [var (name symbol?)])
```

How does this change affect the evaluator?

We need to add a case to handle pow and a case to handle var.