Roadmap

Past three weeks:
• How do we use a functional language?

Next three weeks:
• How do we implement a functional language?
  • ... in a functional language (of course)

This week: Interpreter
• How do we evaluate a program given its abstract syntax tree (AST)?
• How do we prove properties about our interpreter (e.g., that certain programs never crash)?

The Nano Language

Features of Nano:
1. Arithmetic expressions
2. Variables and let-bindings
3. Functions
4. Recursion
Reminder: Calculator

Arithmetic expressions:

\[ e ::= n \]

| e1 + e2 |
| e1 - e2 |
| e1 * e2 |

Example:

\[ 4 + 13 \]

\[ ==> 17 \]

Reminder: Calculator

Haskell datatype to represent arithmetic expressions:

\[
\text{data } \text{Expr} = \text{Num } \text{Int} \\
| \text{Add } \text{Expr} \text{Expr} \\
| \text{Sub } \text{Expr} \text{Expr} \\
| \text{Mul } \text{Expr} \text{Expr}
\]

Haskell function to evaluate an expression:

\[
\text{eval } :: \text{Expr} \rightarrow \text{Int} \\
\text{eval } (\text{Num } n) = n \\
\text{eval } (\text{Add } e1 \text{ e2}) = \text{eval } e1 + \text{eval } e2 \\
\text{eval } (\text{Sub } e1 \text{ e2}) = \text{eval } e1 - \text{eval } e2 \\
\text{eval } (\text{Mul } e1 \text{ e2}) = \text{eval } e1 \times \text{eval } e2
\]

Reminder: Calculator

Alternative representation:

\[
\text{data } \text{Binop} = \text{Add} | \text{Sub} | \text{Mul}
\]

\[
\text{data } \text{Expr} = \text{Num } \text{Int} \quad \text{-- number} \\
| \text{Bin } \text{Binop } \text{Expr} \text{Expr} \quad \text{-- binary expression}
\]

Evaluator for alternative representation:

\[
\text{eval } :: \text{Expr} \rightarrow \text{Int} \\
\text{eval } (\text{Num } n) = n \\
\text{eval } (\text{Bin } \text{Add } e1 \text{ e2}) = \text{eval } e1 + \text{eval } e2 \\
\text{eval } (\text{Bin } \text{Sub } e1 \text{ e2}) = \text{eval } e1 - \text{eval } e2 \\
\text{eval } (\text{Bin } \text{Mul } e1 \text{ e2}) = \text{eval } e1 \times \text{eval } e2
\]
The Nano Language

Features of Nano:
1. Arithmetic expressions [done]
2. Variables and let-bindings
3. Functions
4. Recursion

Extension: variables

Let's add variables and let bindings!

\[ e ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid \text{let } x = e_1 \text{ in } e_2 \]

Example:
\[
\begin{align*}
\text{let } x &= 4 + 13 \text{ in } -- 17 \\
\text{let } y &= 7 - 5 \text{ in } -- 2 \\
x \times y \\
\Rightarrow 34
\end{align*}
\]

Haskell representation:

\[
\text{data } \text{Expr} = \text{Num } \text{Int} \quad -- \text{number} \\
\mid \text{??} \quad -- \text{variable} \\
\mid \text{Bin } \text{Binop } \text{Expr} \text{ Expr} \quad -- \text{binary expression} \\
\mid \text{??} \quad -- \text{let expression}
\]
Extension: variables

type Id = String

data Expr = Num Int -- number
  | Var Id -- variable
  | Bin Binop Expr Expr -- binary expression
  | Let Id Expr Expr -- let expression

Haskell function to evaluate an expression:

eval :: Expr -> Int
eval (Num n) = n
eval (Var x) = ???
...

Environment

An expression is evaluated in an environment, which maps all its free variables to values

Examples:

- How should we represent the environment?
- Which operations does it support?

```haskell```
x * y
= [x:17, y:2] => 34

x * y
= [x:17] => Error: unbound variable y

x * (let y = 2 in y)
= [x:17] => 34````
Extension: variables

What does this evaluate to? *

```
let x = 5 in
let y = x + z in
let z = 10 in

y
```

- (A) 15
- (B) 5
- (C) Error: unbound variable x
- (D) Error: unbound variable y
- (E) Error: unbound variable z

http://tiny.cc/cmps112-vars-ind

---

Environment: API

To evaluate `let x = e1 in e2 in env`:
- evaluate `e2` in an extended environment `env + [x:v]`
- where `v` is the result of evaluating `e1`

To evaluate `x` in `env`:
- lookup the most recently added binding for `x`

```
type Value = Int

data Env = ... -- representation not that important

-- | Add a new binding
add :: Id -> Value -> Env -> Env

-- | Lookup the most recently added binding
lookup :: Id -> Env -> Value
```
Evaluating expressions

Back to our expressions… now with environments!

```
data Expr = Num Int -- number
            | Var Id   -- variable
            | Bin Binop Expr Expr -- binary expression
            | Let Id Expr Expr -- let expression
```

Haskell function to evaluate an expression:

```
eval :: Env -> Expr -> Value

eval env (Num n)   = n
eval env (Var x)   = lookup x env
eval env (Bin op e1 e2) = f v1 v2
  where
    v1 = eval env e1
    v2 = eval env e2
    f = case op of
      Add -> (+)
      Sub -> (-)
      Mul -> (*)
eval env (Let x e1 e2) = eval env' e2
  where
    v  = eval env e1
    env' = add x v env
```

Example evaluation

Nano expression

```
let x = 1 in
let y = (let x = 2 in x) + x in
let x = 3 in
x + y
```

is represented in Haskell as:

```
exp1 = Let "x" (Num 1)
      (Let "y" (Add (Let "x" (Num 2) (Var x)))
        (Var x)))
      (Let "x" (Add (Var x) (Var y)))
```

Example evaluation

eval [] exp1
=> eval [] (let "x" (Num 1) exp2)
=> eval (\ x ,eval [] (Num 1)) exp2
=> eval (\ x ,
      (let "y" (Add exp3 exp4) exp5)
exp5
=> eval (\ y ,exp5) (\ x ,exp5)) (\ x ,1)
exp5
=> eval (\ y ,exp5) (\ x ,1) (Var "x") -- new binding for x
=> eval (\ y ,exp5) (\ x ,1) (Var "x")
ex5
=> eval (\ y ,exp5) (\ x ,1) (Var "x")
ex5
=> eval (\ y ,exp5) (\ x ,1) (Var "x")
ex5
=> eval (\ y ,exp5) (\ x ,1) (Var "x")
ex5
=> eval (\ y ,exp5) (\ x ,1) (Var "x")
ex5

Example evaluation

=> eval (\ y ,2) (\ x ,1) (\ y ,1) (\ y ,1) (Var "y") -- new binding for x
=> eval (\ y ,2) (\ x ,1) (Var "y")
ex5
=> eval (\ y ,2) (\ x ,1) (Var "y")
ex5
=> eval (\ y ,2) (\ x ,1) (Var "y")
ex5
=> eval (\ y ,2) (\ x ,1) (Var "y")
ex5
=> eval (\ y ,2) (\ x ,1) (Var "y")
ex5

Example evaluation

Same evaluation in a simplified format (Haskell Expr terms replaced by their “pretty-
printed version”):

eval []
(let x = 1 in let y = (let x = 2 in x) + x in let x = 3 in x + y)
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5
=> eval [x:eval []] (let y = (let x = 2 in x) + x in let x = 3 in x + y)
ex5

Example evaluation

\[ \text{eval } [y: (2 + 1), x:1] \]
\[ \text{eval } [y:3, x:1] \]
\[ \text{let } x = 3 \text{ in } x + y \]
\[ \text{let } x = 3 \text{ in } x + y \]
\[ \text{-- new binding for } x: \]
\[ \text{eval } [x:3, y:3, x:1] \]
\[ x + y \]
\[ \text{-- use latest binding for } x: \]
\[ 3 + 3 \]
\[ 6 \]

Runtime errors

Haskell function to evaluate an expression:
\[
\text{eval} :: \text{Env} \to \text{Expr} \to \text{Value}
\]
\[
\text{eval } \text{env} (\text{Num } n) = n
\]
\[
\text{eval } \text{env} (\text{Var } x) = \text{lookup } x \text{ env} \quad \text{-- can fail!}
\]
\[
\text{eval } \text{env} (\text{Bin } \text{op } e1 e2) = f \; v1 \; v2
\]
where
\[
v1 = \text{eval } \text{env} e1
\]
\[
v2 = \text{eval } \text{env} e2
\]
\[
f = \text{case } \text{op} \text{ of}
\]
\[
\text{Add } \to (+)
\]
\[
\text{Sub } \to (-)
\]
\[
\text{Mul } \to (*)
\]
\[
\text{eval } \text{env} (\text{Let } x \; e1 \; e2) = \text{eval } \text{env'} e2
\]
where
\[
v = \text{eval } \text{env} e1
\]
\[
\text{env'} = \text{add } x \; v \; \text{env}
\]

How do we make sure lookup doesn’t cause a run-time error?

Free vs bound variables

In \text{eval } \text{env } e, \text{env} must contain bindings for all free variables of \text{e}!

- an occurrence of \text{x} is free if it is not bound
- an occurrence of \text{x} is bound if it’s inside \text{e2} where \text{let } x = e1 \text{ in } e2
- evaluation succeeds when an expression is closed!
The Nano Language

Features of Nano:
1. Arithmetic expressions [done]
2. Variables and let-bindings [done]
3. Functions
4. Recursion
Extension: functions

Let's add lambda abstraction and function application!

\[
e ::= n | x
| e1 + e2 | e1 - e2 | e1 \times e2
| \text{let } x = e1 \text{ in } e2
| \lambda x \to e -- \text{ abstraction}
| e1 \ e2 -- \text{ application}
\]

Example:

\[
\text{let } c = 42 \text{ in}
\text{let } \text{cTimes} = \lambda x \to c \times x \text{ in}
\text{cTimes} \ 2
\\[\Rightarrow 84\]

Extension: functions

Haskell representation:

```
data Expr = Num Int      -- number
    | Var Id           -- variable
    | Bin Binop Expr Expr  -- binary expression
    | Let Id Expr Expr   -- let expression
    | ???               -- abstraction
    | ???               -- application
```

Extension: functions

Haskell representation:

```
data Expr = Num Int      -- number
    | Var Id           -- variable
    | Bin Binop Expr Expr  -- binary expression
    | Let Id Expr Expr   -- let expression
    | Lam Id Expr       -- abstraction
    | App Expr Expr     -- application
```
Extension: functions

Example:

```plaintext
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
```

represented as:

```plaintext
Let "c"
  (Num 42)
(Let "cTimes"
  (Lam "x" (Mul (Var "c") (Var "x")))))
(App (Var "cTimes") (Num 2)))
```

---

Extension: functions

Example:

```plaintext
let c = 42 in
let cTimes = \x -> c * x in
cTimes 2
```

How should we evaluate this expression?

```plaintext
eval []
  {let c = 42 in let cTimes = \x -> c * x in cTimes 2}
=> eval [c:42]
=> eval [cTimes:???, c:42]
    {cTimes 2}
```

What is the value of cTimes??

---

Rethinking our values

Until now: a program evaluates to an integer (or fails)

```plaintext
type Value = Int
type Env = [(Id, Value)]
eval :: Env -> Expr -> Value
```
Rethinking our values

What do these programs evaluate to?

(1)
\[ x \rightarrow 2 \times x \]
\[ \Rightarrow ??? \]

(2)
let \( f = \lambda x \rightarrow \lambda y \rightarrow 2 \times (x + y) \) in
\[ f \ 5 \]
\[ \Rightarrow ??? \]

Conceptually, (1) evaluates to itself (not exactly, see later). while (2) evaluates to something equivalent to \( \lambda y \rightarrow 2 \times (5 + y) \)

---

Rethinking our values

Now: a program evaluates to an integer or a lambda abstraction (or fails)

• Remember: functions are first-class values

Let's change our definition of values!

data Value = VNum Int |

VLam ???

-- What info do we need to store?

-- Other types stay the same
type Env = [(Id, Value)]

eval :: Env -> Expr -> Value

---

Function values

How should we represent a function value?

let \( c = 42 \) in
let \( \text{cTimes} = \lambda x \rightarrow c \times x \) in
\( \text{cTimes} \ 2 \)

We need to store enough information about \( \text{cTimes} \) so that we can later evaluate any application of \( \text{cTimes} \) (like \( \text{cTimes} \ 2 \))!

First attempt:

data Value = VNum Int |

VLam Id Expr -- formal + body
### Function values

Let's try this!

```
let c = 42 in let cTimes = \x -> c * x in cTimes 2
```

```
=> eval [c:42] {let cTimes = \x -> c * x in cTimes 2}
```

```
=> eval [cTimes:(\x -> c*x), c:42] {cTimes 2}
```

```
-- evaluate the function:
```

```
=> eval [cTimes:(\x -> c*x), c:42] {\x -> c * x} 2
```

```
-- evaluate the argument, bind to x, evaluate body:
```

```
=> eval [x:2, cTimes:(\x -> c*x), c:42] {c * x}
```

```
=> 42 * 2
```

```
=> 84
```

Looks good… can you spot a problem?

---

### QUIZ

What should this evaluate to? *

```
let c = 42 in
let cTimes = \x -> c * x
```

---

http://tiny.cc/cmps112-cscope-ind

---

### QUIZ

What should this evaluate to? *

```
let c = 42 in
let cTimes = \x -> c * x
```

---

http://tiny.cc/cmps112-cscope-grp
Static vs Dynamic Scoping

What we want:

```plaintext
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 84
```

Lexical (or static) scoping:
- each occurrence of a variable refers to the most recent binding in the program text
- definition of each variable is unique and known statically
- good for readability and debugging: don’t have to figure out where a variable got “assigned”

Static vs Dynamic Scoping

What we don’t want:

```plaintext
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 10
```

Dynamic scoping:
- each occurrence of a variable refers to the most recent binding during program execution
- can’t tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```plaintext
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!!
res2 - res1
```
Function values

```
data Value = VNum Int 
| VLam Id Expr -- formal + body

This representation can only implement dynamic scoping!

let c = 42 in 
let cTimes = \x -> c * x in 
let c = 5 in 
cTimes 2
```
evaluates as:
```
eval []
{let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
```

```
eval []
{let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
=> eval [c:42]
=> eval [cTimes:{\x -> c*x}, c:42]
=> eval [c:5, cTimes:{\x -> c*x}, c:42]
=> eval [x:2, c:5, cTimes:{\x -> c*x}, c:42]
{\x -> c * x}
-- latest binding for c is 5!
=> 5 * 2
=> 10
```

Lesson learned: need to remember what c was bound to when cTimes was defined!

- i.e. “freeze” the environment at function definition

Closures

To implement lexical scoping, we will represent function values as closures

```
Closure = lambda abstraction (formal + body) + environment at function definition

data Value = VNum Int 
| VClos Env Id Expr -- env + formal + body
```
Closures

Our example:

```haskell
eval []
  {let c = 42 in let cTimes = \x -> c * x in let c = 5 in cTimes 2}
-> eval [c:42]
  {let cTimes = \x -> c * x in let c = 5 in cTimes 2}
  -- remember current env:
-> eval [cTimes:{c:42}, \x -> c*x}, c:42]
  {let c = 5 in cTimes 2}
-> eval [c:5, cTimes:{c:42}, \x -> c*x}, c:42]
  {cTimes 2}
-> eval [c:5, cTimes:{c:42}, \x -> c*x}, c:42]
  {({c:42}, \x -> c * x) 2}
  -- restore env to the one inside the closure, then bind 2 to x:
-> eval [x:2, c:42]
  {c + x}
->
  42 * 2
->
  84
```

Free vs bound variables

- An occurrence of x is free if it is not bound
- An occurrence of x is bound if it's inside
  - e2 where let x = e1 in e2
  - e where \x -> e
- A closure environment has to save all free variables of a function definition!

```haskell
let a = 20 in
let f =
  \x -> let y = x + 1 in
  let g = \z -> y + z in
  a + g x -- a is the only free variable!
```

Evaluator

Let's modify our evaluator to handle functions!

```haskell
data Value = VNum Int
            | VClos Env Id Expr -- env + formal + body
eval :: Env -> Expr -> Value
eval env (Num n) = VNum n -- must wrap in VNum now!
eval env (Var x) = lookup x env
eval env (Bin op e1 e2) = VNum (f v1 v2)
  where
    (VNum v1) = eval env e1
    (VNum v2) = eval env e2
    f = ... -- as before
  eval env (Let x e1 e2) = eval env' e2
  where
    v = eval env e1
    env' = add x v env
eval env (Lam x body) = ??? -- construct a closure
eval env (App fun arg) = ??? -- eval fun, then arg, then apply
```
Evaluator

Evaluating functions:

- Construct a closure: save environment at function definition
- Apply a closure: restore saved environment, add formal, evaluate the body

```
eval :: Env -> Expr -> Value
...  
eval env (Lam x body) = VClos env x body
   where (VClos closEnv x body) = eval env fun -- eval function to closure
        vArg = eval env arg -- eval argument
        bodyEnv = add x vArg closEnv
```

Lesson learned: to support recursion, we need a different way of constructing the closure environment!
Formalizing Nano

Goal: we want to guarantee properties about programs, such as:
• evaluation is deterministic
• all programs terminate
• certain programs never fail at run time
• etc.

To prove theorems about programs we first need to define formally:
• their syntax (what programs look like)
• their semantics (what it means to run a program)

Let’s start with Nano1 (Nano w/o functions) and prove some stuff!

Nano1: Syntax

We need to define the syntax for expressions (terms) and values using a grammar:

\[
\begin{align*}
e & ::= n \mid x \quad \text{-- expressions} \\
& \quad \mid e_1 + e_2 \\
& \quad \mid \text{let } x = e_1 \text{ in } e_2 \\
v & ::= n \quad \text{-- values} \\
\end{align*}
\]

where \( n \in \mathbb{N}, x \in \text{Var} \)

Nano1: Operational Semantics

Operational semantics defines how to execute a program step by step.

Let’s define a step relation (reduction relation) \( e \Rightarrow e' \):
• “expression \( e \) makes a step (reduces in one step) to an expression \( e' \)”
Nano1: Operational Semantics

We define the step relation inductively through a set of rules:

**[Add-L]**

\[ e1 \Rightarrow e1' \]  \hspace{2cm} \text{-- premise}

\[ e1 + e2 \Rightarrow e1' + e2 \]  \hspace{2cm} \text{-- conclusion}

**[Add-R]**

\[ e2 \Rightarrow e2' \]

**[Add]**

\[ n1 + n2 \Rightarrow n \hspace{2cm} \text{where } n = n1 + n2 \]

**[Let-Def]**

\[ \text{let } x = e1 \text{ in } e2 \Rightarrow \text{let } x = e1' \text{ in } e2 \]

**[Let]**

\[ \text{let } x = v \text{ in } e2 \Rightarrow e2[x := v] \]

---

Nano1: Operational Semantics

Here \( e[x := v] \) is a value substitution:

\[ x[x := v] = v \]
\[ y[x := v] = y \hspace{2cm} \text{-- assuming } x \neq y \]
\[ n[x := v] = n \]
\[ (e1 + e2)[x := v] = e1[x := v] + e2[x := v] \]
\[ (\text{let } x = e1 \text{ in } e2)[x := v] = \text{let } x = e1[x := v] \text{ in } e2 \]
\[ (\text{let } y = e1 \text{ in } e2)[x := v] = \text{let } y = e1[x := v] \text{ in } e2 \]
\[ e2[x := v] \]

Do not have to worry about capture, because \( v \) is a value (has no free variables!)

---

Nano1: Operational Semantics

A reduction is valid if we can build its derivation by “stacking” the rules:

**[Add]**

\[ 1 + 2 \Rightarrow 3 \]

**[Add-L]**

\[ (1 + 2) + 5 \Rightarrow 3 + 5 \]

Do we have rules for all kinds of expressions?
Nano1: Operational Semantics

We define the step relation inductively through a set of rules:

\[ \text{Add-L} \quad \frac{}{e_1 \Rightarrow e_1'} \quad \text{-- premise} \]
\[ e_1 + e_2 \Rightarrow e_1' + e_2 \quad \text{-- conclusion} \]
\[ e_2 \Rightarrow e_2' \]

\[ \text{Add-R} \quad \frac{}{n_1 + e_2 \Rightarrow n_1 + e_2'} \]
\[ n_1 + n_2 \Rightarrow n \quad \text{where } n = n_1 + n_2 \]

\[ \text{Add} \]
\[ \text{Let-Def} \]
\[ \text{Let} \]

1. Normal forms

There are no reduction rules for:
- \( n \)
- \( x \)

Both of these expressions are normal forms (cannot be further reduced), however:
- \( n \) is a value
  - Intuitively, corresponds to successful evaluation
- \( x \) is not a value
  - Intuitively, corresponds to a run-time error!
  - We say the program \( x \) is stuck

2. Evaluation order

In \( e_1 + e_2 \), which side should we evaluate first?

In other words, which one of these reductions is valid (or both)?
1. \( (1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5) \)
2. \( (1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9 \)

Reduction (1) is valid because we can build a derivation using the rules:

\[ \text{Add} \]
\[ \frac{}{1 + 2 \Rightarrow 3} \]
\[ \frac{}{(1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5)} \]

Reduction (2) is invalid because we cannot build a derivation:
- There is no rule whose conclusion matches this reduction!

\[ \frac{}{(1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9} \]
**Evaluation relation**

Like in λ-calculus, we define the multi-step reduction relation $e \Rightarrow e'$:

$$e \Rightarrow e' \iff$$

- $e = e_1$
- $e_n = e'$
- $e_i \Rightarrow e_{i+1}$ for each $i$ in $[0..n)$

Example:

$$(1 + 2) + (4 + 5) \Rightarrow 3 + 9$$

because

$$(1 + 2) + (4 + 5)$$

$$\Rightarrow 3 + (4 + 5)$$

$$\Rightarrow 3 + 9$$

$$\Rightarrow 12$$

**Evaluation relation**

Now we define the evaluation relation $e \Rightarrow\Rightarrow e'$:

$$e \Rightarrow\Rightarrow e' \iff$$

- $e \Rightarrow e'$
- $e'$ is in normal form

Example:

$$(1 + 2) + (4 + 5) \Rightarrow\Rightarrow 12$$

because

$$(1 + 2) + (4 + 5)$$

$$\Rightarrow 3 + (4 + 5)$$

$$\Rightarrow 3 + 9$$

$$\Rightarrow 12$$

and $12$ is a value (normal form)

**Theorems about Nano1**

Let’s prove something about Nano1!

1. Every Nano1 program terminates
2. Closed Nano1 programs don’t get stuck
3. Corollary (1 + 2): Every closed Nano1 program evaluates to a value

How do we prove theorems about languages?

**By induction.**
1. Induction on natural numbers

To prove $\forall n. P(n)$ we need to prove:

- **Base case**: $P(0)$
- **Inductive case**: $P(n+1)$ assuming the induction hypothesis (IH): that $P(n)$ holds

Compare with inductive definition for natural numbers:

```hs
data Nat = Zero -- base case
         | Succ Nat -- inductive case
```

No reason why this would only work for natural numbers...

In fact we can do induction on any inductively defined mathematical object (= any datatype)!

- lists
- trees
- programs (terms)
- etc

2. Induction on terms

$$e ::= n \mid x$$

$$\mid e_1 + e_2$$

$$\mid \text{let } x = e_1 \text{ in } e_2$$

To prove $\forall e. P(e)$ we need to prove:

- **Base case 1**: $P(n)$
- **Base case 2**: $P(x)$
- **Inductive case 1**: $P(e_1 + e_2)$ assuming the IH:
  - that $P(e_1)$ and $P(e_2)$ hold
- **Inductive case 2**: $P(\text{let } x = e_1 \text{ in } e_2)$ assuming the IH:
  - that $P(e_1)$ and $P(e_2)$ hold
3. Induction on derivations

Our reduction relation $\Rightarrow$ is also defined **inductively**!

- Axioms are base cases
- Rules with premises are inductive cases

To prove $\forall e, e' \cdot P(e \Rightarrow e')$ we need to prove:

- **Base cases:** [Add], [Let]
- **Inductive cases:** [Add-L], [Add-R], [Let-Def] assuming the IH: that $P$ holds of their premise

---

**Theorem: Termination**

**Theorem 1 [Termination]:** For any expression $e$ there exists $e'$ such that $e \Rightarrow e'$.

Proof idea: let's define the size of an expression such that

- size of each expression is positive
- each reduction step strictly decreases the size

Then the length of the execution sequence for $e$ is bounded by the size of $e$!

\[
\begin{align*}
\text{size } n &= \ ??
\\
\text{size } x &= \ ??
\\
\text{size } (e_1 + e_1) &= \ ??
\\
\text{size } (\text{let } x = e_1 \text{ in } e_2) &= \ ??
\end{align*}
\]

---

**Theorem: Termination**

Term size:

\[
\begin{align*}
\text{size } n &= 1 \\
\text{size } x &= 1 \\
\text{size } (e_1 + e_1) &= \text{size } e_1 + \text{size } e_2 \\
\text{size } (\text{let } x = e_1 \text{ in } e_2) &= \text{size } e_1 + \text{size } e_2
\end{align*}
\]

**Lemma 1:** For any $e$, size $e > 0$.

Proof: By induction on the term $e$.

- **Base case 1:** size $n = 1 > 0$
- **Base case 2:** size $x = 1 > 0$
- **Inductive case 1:** size $(e_1 + e_2) = \text{size } e_1 + \text{size } e_2 > 0$ because size $e_1 > 0$ and size $e_2 > 0$ by IH.
- **Inductive case 2:** similar.

QED.
Theorem: Termination

Lemma 2: For any \( e, e' \) such that \( e \rightarrow e' \), size \( e' \) < size \( e \).

Proof: By induction on the derivation of \( e \rightarrow e' \).

Base case [Add].

• Given: the root of the derivation is
  
  \[ \text{Add}: n_1 + n_2 \rightarrow n \text{ where } n = n_1 + n_2 \]

• To prove: size \( n \) < size \( (n_1 + n_2) \)
  
  size \( n = 1 < 2 = \) size \( (n_1 + n_2) \)

Inductive case [Add-L].

• Given: the root of the derivation is [Add-L]:
  
  \[ e_1 \rightarrow e_1' \]

  \[ e_1 + e_2 \rightarrow e_1' + e_2 \]

• To prove: size \( (e_1' + e_2) \) < size \( (e_1 + e_2) \)
  
  IH: size \( e_1' \) < size \( e_1 \)

  size \( (e_1' + e_2) \)
  \[ = \text{-- def. size} \]
  
  size \( e_1' + e_2 \)
  \[ < \text{-- IH} \]
  
  size \( e_1 + e_2 \)
  \[ = \text{-- def. size} \]
  
  size \( e_1 + e_2 \)

Inductive case [Add-R]. Try at home

Base case [Let].

• Given: the root of the derivation is
  
  \[ \text{Let}: \text{let } x = v \text{ in } e_2 \rightarrow e_2[x := v] \]

• To prove: size \( e_2[x := v] \) < size \( \text{let } x = v \text{ in } e_2 \)
  
  size \( e_2[x := v] \)
  \[ = \text{-- auxiliary lemma} \]
  
  size \( e_2 \)
  \[ < \text{-- IH} \]
  
  size \( v + e_2 \)
  \[ = \text{-- def. size} \]
  
  size \( \text{let } x = v \text{ in } e_2 \)

QED.
Nano2: adding functions

Syntax

We need to extend the syntax of expressions and values:

\[
\begin{align*}
e ::= & \ n \ | \ x \quad \text{-- expressions} \\
& \ e_1 + e_2 \\
& \ \text{let } x = e_1 \ \text{in } e_2 \\
& \ \lambda x \to e \quad \text{-- abstraction} \\
& \ e_1 \ e_2 \quad \text{-- application} \\
\end{align*}
\]

\[
\begin{align*}
v ::= & \ n \quad \text{-- values} \\
& \ \lambda x \to e \quad \text{-- abstraction} \\
\end{align*}
\]

Operational semantics

We need to extend our reduction relation with rules for abstraction and application:

\[
\begin{align*}
e_1 \to e_1' \\
[\text{App-L}] & \quad e_1 e_2 \to e_1' e_2 \\
& \quad \ e \to e' \\
[\text{App-R}] & \quad \nu \ n \to \nu \ e' \\
& \quad \ (\lambda x \to e) \ \nu \to e[x := \nu]
\end{align*}
\]
Evaluation Order

\[ ((\lambda y \rightarrow x + y) \ 1 \ 1 + 2) \]
\[ \Rightarrow (\lambda y \rightarrow 1 + y) \ 1 + 2 \ -- \ [\text{App-L}], \ [\text{App}] \]
\[ \Rightarrow (\lambda y \rightarrow 1 + y) \ 3 \ -- \ [\text{App-R}], \ [\text{Add}] \]
\[ \Rightarrow 1 + 3 \ -- \ [\text{Add}] \]
\[ \Rightarrow 4 \ -- \ [\text{Add}] \]

Our rules define call-by-value:

1. Evaluate the function (to a lambda)
2. Evaluate the argument (to some value)
3. “Make the call”: make a substitution of formal to actual in the body of the lambda

The alternative is call-by-name:

- do not evaluate the argument before “making the call”
- can we modify the application rules for Nano2 to make it call-by-name?

Theorems about Nano2

Let’s prove something about Nano2!

1. Every Nano2 program terminates (?)
2. Closed Nano2 programs don’t get stuck (?)

Theorems about Nano2

1. Every Nano2 program terminates (?)

   What about \((\lambda x \rightarrow x) \ (\lambda x \rightarrow x)\)?

2. Closed Nano2 programs don’t get stuck (?)

   What about \(1 + 2\)?

Both theorems are now false!
To recover these properties, we need to add types:

1. Every well-typed Nano2 program terminates

2. Well-typed Nano2 programs don’t get stuck

We’ll do that next week!