Environments

Past three weeks

How to use essential language constructs?

- Data Types
- Recursion
- Higher-Order Functions

Next two weeks

How to implement language constructs?

- Local variables and scope
- Environments and Closures
- (skip) Type Inference

Interpreter

How do we represent and evaluate a program?
Roadmap: The Nano Language

Features of Nano:

1. Arithmetic
2. Variables
3. Let-bindings
4. Functions
5. Recursion

1. Nano: Arithmetic

A “grammar” of arithmetic expressions:
\[ e ::= n \]

\[ | e_1 + e_2 \]

\[ | e_1 - e_2 \]

\[ | e_1 \ast e_2 \]

(A) have heard of (B) huh?

### Expressions vs. Values

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>==&gt; 4</td>
</tr>
<tr>
<td>4 + 12</td>
<td>==&gt; 16</td>
</tr>
<tr>
<td>(4+12) - 5</td>
<td>==&gt; 11</td>
</tr>
</tbody>
</table>

#### Representing Arithmetic Expressions and Values

![Diagram: Expressions to Values]

Let's represent arithmetic expressions as type...
data Expr
    = ENum Int -- ^ n
    | EAdd Expr Expr -- ^ e1 + e2
    | ESub Expr Expr -- ^ e1 - e2
    | EMul Expr Expr -- ^ e1 * e2

Lets represent arithmetic values as a type

type Value = Int

data Value = VInt Int

Evaluating Arithmetic Expressions

We can now write a Haskell function to evaluate an expression:

eval :: Expr -> Value
eval (ENum n) = n
eval (EAdd e1 e2) = eval e1 + eval e2
eval (ESub e1 e2) = eval e1 - eval e2
eval (EMul e1 e2) = eval e1 * eval e2
Alternative representation

Lets pull the operators into a separate type

```haskell
data Binop = Add -- ^ `+`
  | Sub -- ^ `-
  | Mul -- ^ `*`
...

data Expr = ENum Int -- ^ n
  | EBin Binop Expr Expr -- ^ e1 `op` e2
```

QUIZ

Evaluator for alternative representation
eval :: Expr -> Value

\[
\text{eval (ENum } n) = n
\]

\[
\text{eval (EBin } \text{op } e1 \text{ e2) = evalOp op (eval e1) (eval e2)}
\]

What is a suitable type for evalOp?

{- 1 -} evalOp :: BinOp -> Value

{- 2 -} evalOp :: BinOp -> Value -> Value -> Value

{- 3 -} evalOp :: BinOp -> Expr -> Expr -> Value

{- 4 -} evalOp :: BinOp -> Expr -> Expr -> Expr

{- 5 -} evalOp :: BinOp -> Expr -> Value

The Nano Language

Features of Nano:

1. Arithmetic [done]
2. Variables
3. Let-bindings
4. Functions
5. Recursion
2. Nano: Variables

Let’s add variables and let bindings!

\[
e ::= n \quad -- \text{OLD}
\]

\[
| e1 + e2 \quad \text{old}
| e1 - e2 \quad \text{old}
| e1 * e2
\]

-- NEW

\[
| x \quad -- \text{variables}
\]

\[(10+4) - 5\]

\[(10+a) \times (b-c)\]

Lets extend our datatype
type Id = String

data Expr
  = EEnum Int  -- OLD
       | EBin Binop Expr Expr  -- NEW
       | EVar Id  -- variables

**QUIZ**

What should the following expression evaluate to?

\[ x + 1 \]

(A) 0

(B) 1

(C) Error

Environment
An expression is evaluated in an environment

- A phone book which maps variables to values

\[
\begin{array}{c}
"x" : = 0, "y" : = 12, \ldots
\end{array}
\]

A type for environments

\texttt{type Env} = [(Id, Value)]

---

**Evaluation in an Environment**

We write

\[(\text{eval env expr}) \Rightarrow value\]

to mean

When \texttt{expr} is evaluated in environment \texttt{env} the result is \texttt{value}

That is, when we have variables, we modify our evaluator to take an input environment \texttt{env} in which \texttt{expr} must be evaluated.

\texttt{eval :: Env \rightarrow Expr \rightarrow Value}

\texttt{eval env expr = ... value-of-expr-in-env...}

First, let's update the evaluator for the arithmetic cases \texttt{ENum} and \texttt{EBin}
eval :: Env -> Expr -> Value

eval env (ENum n) = ???

eval env (EBin op e1 e2) = ???
What is a suitable env such that

\[
\text{eval env} (x + 1) \implies 10
\]

\(\times\) (A) \(\left[\right]\) \(\rightarrow\) \text{undefined}

\(\times\) (B) \([x := 0, y := 9]\) \(\rightarrow\) 1

✓ (C) \([x := 9, y := 0]\)

✓ (D) \([x := 9, y := 10, z := 666]\)

✓ (E) \([y := 10, z := 666, x := 9]\)

---

**Evaluating Variables**

Using the above intuition, let's update our evaluator to handle variables i.e. the EVar case:

\[
\text{eval env} (\text{EVar} \ x) = ???
\]

Let's confirm that our eval is ok!
envA = []
envB = ["x" := 0 , "y" := 9]
envC = ["x" := 9 , "y" := 0]
envD = ["x" := 9 , "y" := 10 , "z" := 666]
envE = ["y" := 10 , "z" := 666, "x" := 9 ]

-- >>> eval envA (EBin Add (EVar "x") (ENum 1))
-- >>> eval envB (EBin Add (EVar "x") (ENum 1))
-- >>> eval envC (EBin Add (EVar "x") (ENum 1))
-- >>> eval envD (EBin Add (EVar "x") (ENum 1))
-- >>> eval envE (EBin Add (EVar "x") (ENum 1))

**The Nano Language**

Features of Nano:

1. Arithmetic expressions [done]
2. Variables [done]
3. Let-bindings
4. Functions
5. Recursion
2. Nano: Variables

Let’s add variables and let bindings!

\[
e ::= n \quad \text{-- OLD}
\]
\[
| e_1 + e_2
| e_1 - e_2
| e_1 \times e_2
| x
\]
\[
| \text{let } x = e_1 \text{ in } e_2 \quad \text{-- NEW}
\]

Let's extend our datatype

```haskell
type Id = String
```

```haskell
data Expr
  = ENum Int \quad \text{-- OLD}
  | EBin Binop Expr Expr
  | EVar Id
  \quad \text{-- NEW}
  | ELet Id Expr Expr
```

How should we extend eval?
**QUIZ**

What **should** the following expression evaluate to?

```
let x = 0
in
  x + 1
```

- (A) Error
- (B) 1
- (C) 0

**QUIZ**

What **should** the following expression evaluate to?

```
let x = e₁,
in
  let y = e₂
  in
    x + y
```

(A) Error
**QUIZ**

What *should* the following expression evaluate to?

```plaintext
let x = 0
in

let x = 100
in
x + 1
```

- (A) Error
- (B) 0
- (C) 1
- (D) 100
- (E) 101