CS 4400 / 5400
Programming Languages

[03: Names, Scope / Environments]

Ferdinand Vesely

September 17, 2019

Recap

Recap

We mentioned concrete syntax...

Concrete Syntax

What does an expression look like?

```
2 + 4
```

```
     Exp
    / | \
   /  |  \
  /   |   \
Exp  '+' Exp
 |         |
 |         |
'2'       '4'
```

Recap

We talked about abstract syntax…

Abstract Syntax

What are the (semantically) significant / essential parts of an expression?

$$2 + 4$$

```
      plus
      /  \
     /    \
    /      \
   2        4
```

Do not worry about the details, what symbols are used to represent operations.

Recap

We talked about BNF…

BNF (Backus-Naur Form)

A formalism for specifying syntax (concrete or abstract).
Concrete:

```
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

<Decimal> ::= <Digit> | <Digit> <Decimal>

<Exp> ::= <Decimal>
        | <Exp> '+' <Exp>
        | '(' <Exp> ')'
```

Recap

BNF (Backus-Naur Form)

Abstract:

```
Assume <Nat>, the natural numbers

<AExpr> ::= <AExpr> + <AExpr>    // addition
          | <Nat>                // number literal
```

In Haskell:

```haskell
type Nat = Integer              -- type synonym for "naturals"

data AExpr = Add AExpr AExpr    -- <AExpr> + <AExpr>
           | Num Nat            -- <Nat>
```

Recap

Haskell	Abstract	Concrete

`Add (Num 1) (Num 2)`	$1+2$	`(+ 1 2)`
		`1 + 2`
		`(1 2 +)`

Recap

We talked about evaluators…

```haskell
eval :: AExpr -> Integer
eval (Add ae1 ae2) = eval ae1 + eval ae2
eval (Num n) = n
```

Recap

We talked about bindings, substitution…

```
let x = 3 in x + 4
```

Today

- More bindings
- On scope
- Environments
- More than one type of value

Note

I will switch to Scheme-like s-expressions for concrete representations or *our* languages.

That is, I will write:

`(+ 10 20)`	instead of	`10 + 20`
`(let (x 30) (+ x x))`	instead of	`let x = 30 in x + x`
etc.		

This is to distinguish our example languages from Haskell.

Bindings

Let bindings

```
(let (x (+ 10 20)) (* x x))
```

> *"Evaluate $10+20$ to a value, then replace all occurrences of x in `(* x x)` with that value. Finally compute the value of that expression.*

```
eval (Let x ae1 ae2) =
  let v1 = eval ae1
      ae2' = subst x v1 ae2
  in eval ae2'
```

We use a helper function, `subst` to do the actual substitution.

Substitution

```
subst :: Vars -> Integer -> AExpr -> AExpr
subst x v (Var y) | x == y = Num v      -- variable found!
                  | x /= Var y          -- not "our" variable
subst _ _ (Num i) = Num i               -- nothing to substitute
subst x v (Add ae1 ae2) = Add (subst x v ae1) (subst x v ae2)
subst x v (Let y ae1 ae2)
  | x == y = Let y (subst x v ae1) ae2  -- peculiar case
  | x /= y = Let y (subst x v ae1) (subst x v ae2)
```

Scopes

```
(let (x 10) (+ x (let (x (+ x 32)) (* 2 x))))
```

```
  let x
 /     \
10      +
       / \
      x   \
         let x
         /    \
        +      *
       / \    / \
      x  32  2   x
```

Environments

Maps between variables and values (or expressions)

- Can be thought of as “lazy” or “delayed” substitution.

Three operations:

1. `empty :: Env a`
   - create an empty environment
2. `add :: Var -> a -> Env a -> Env a`
   - add a binding to an environment
   - sometimes also called update or extend
3. `get :: Var -> Env a -> a`
   - find the value bound to the given variable
   - also called `find`, `lookup`

The type `Env a` = environments binding variables to values of type a

- e.g., `Env Integer`

Environment Axioms

- Different possible implementations
- However, they need to satisfy these axioms:

1. `get x (add x v env) == v`
2. `get x (add y v env) == get x env` if x `/= y`
3. `get x empty` is undefined (results in an error) for any x

Environments

For example:

- Create an environment containing a single binding of `"x"` to the integer 42 (the type of the result will be Env `Integer`)

```
add "x" 42 empty
```

- Find the binding for `"x"` in an environment (applying the axioms):

```
get "x" (add "y" 10 (add "z" 20 (add "x" 30 empty)))
  = get "x" (add "z" 20 (add "x" 30 empty))
  = get "x" (add "x" 30 empty)
  = 30
```