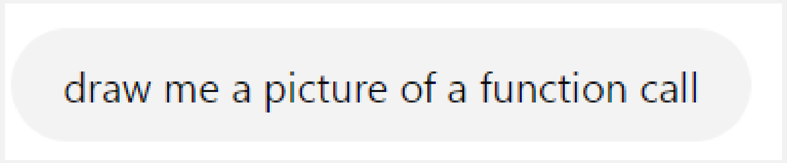
UMass Boston Computer Science

CS450 High Level Languages (section 2)

Function Calls and Functions

Wednesday, November 20, 2024

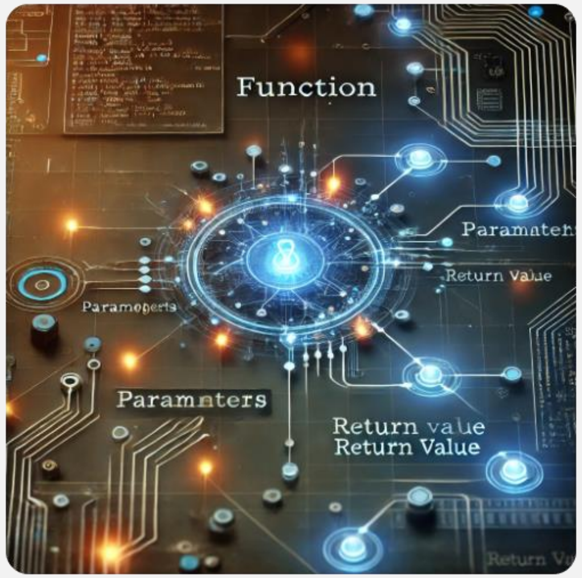


draw me a picture of a function call

Logistics

- HW 11 out
  - due: Mon 11/25 12pm noon EST

- HW 12
  - out: Mon 11/25 12pm noon EST
  - due: Wed 12/4 12pm noon EST



Last Time

The "CS450" Programming Lang! (so far)

```
;; An Atom is:
;; - Number
;; - …
```

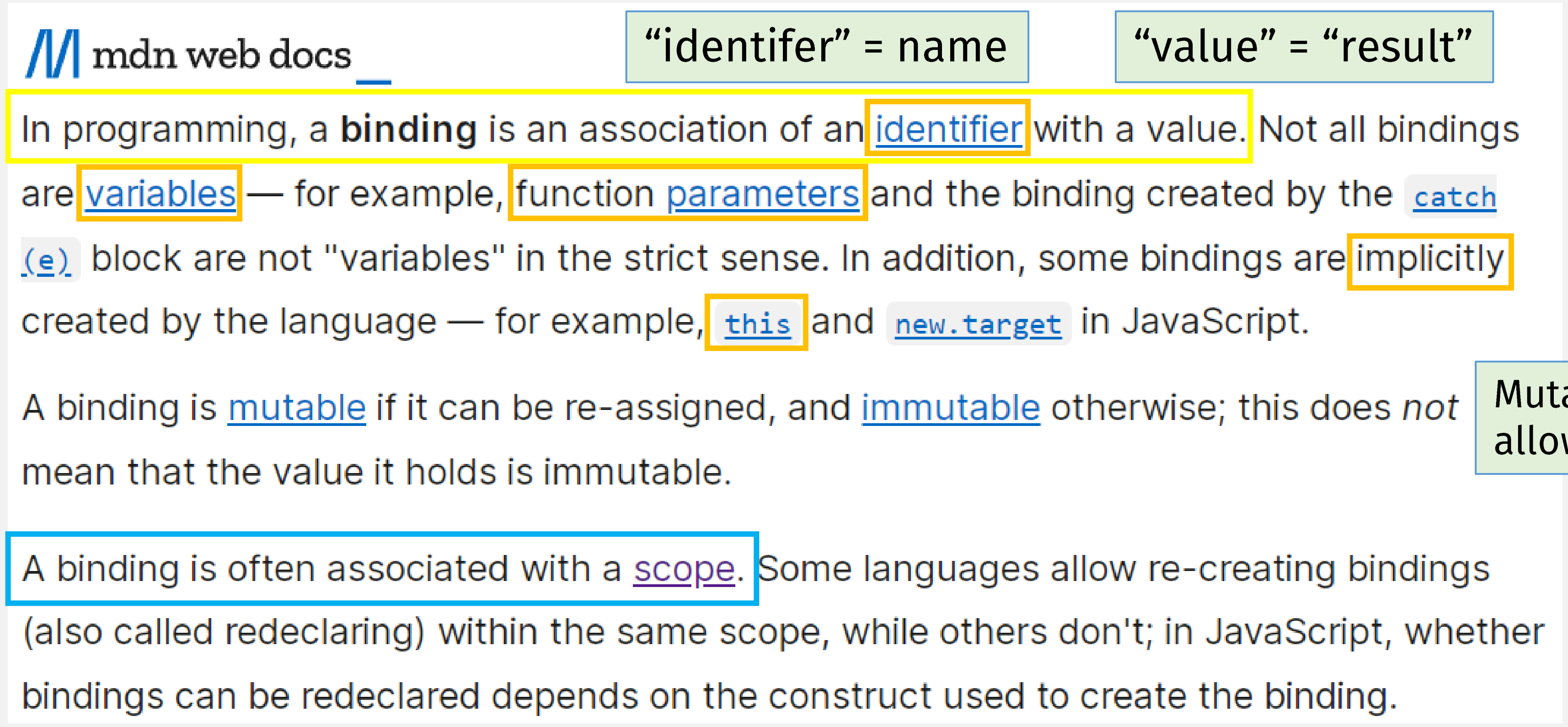
```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - '(bind [Var Expr] Expr)
```

Variable reference

```
;; A Variable (Var) is a Symbol
```

Create new variables

Interlude: What is a “binding”?



mdn web docs

“identifer” = name

“value” = “result”

In programming, a **binding** is an association of an identifier with a value. Not all bindings are variables — for example, function parameters and the binding created by the `catch (e)` block are not "variables" in the strict sense. In addition, some bindings are implicitly created by the language — for example, `this` and `new.target` in JavaScript.

A binding is mutable if it can be re-assigned, and immutable otherwise; this does *not* mean that the value it holds is immutable.

Mutation (e.g., `set!`) not allowed in this class (so far)

A binding is often associated with a scope. Some languages allow re-creating bindings (also called redeclaring) within the same scope, while others don't; in JavaScript, whether bindings can be redeclared depends on the construct used to create the binding.

`https://developer.mozilla.org/en-US/docs/Glossary/Binding`

Bind scoping examples

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
```

bind obeys “**lexical**” or “**static**” scoping

Generally accepted to be “best choice” for programming language design
(bc it’s determined only by program syntax)

Var binding

Var reference

```
(check-equal?
  (eval450 ‘(bind [x 10] x))
  10 ) ; no shadow
```

```
(check-equal?
  (eval450 ‘(bind [x 10]
              (bind [x 20]
                x)))
  20 ) ; shadow
```

```
(check-equal?
  (eval450
    ‘(bind [x 10]
        (+ (bind [x 20] x)
           x)))
  30 )
```

```
(check-equal?
  (eval450
   ‘(bind [x 10]
      (bind [x (+ x 20)]
        x)))
   30 )
```

Last Time

The “CS450” Programming Lang! (so far)

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
```

Last Time

The “CS450” Programming Lang! (so far)

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
```

parse

```
;; An AST is one of:
;; - …
;; - (vari Symbol)
;; - (bind Symbol AST AST)

;; …
(struct vari [name])
(struct bind [var expr body])
```

“eval450”

```
;; A Result is a:
;; - Number
;; - …
```

run

(JS semantics)

run needs an **accumulator** to “remember” variables and their `Results`

run with an Environment accumulator

Environment has **Results** (not **AST**)

Need to run Exprs before adding to env!

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; interp: a runtime environment
;; for cs450-lang vars; same-name
;; vars in front shadow later ones
```

```
;; run: AST -> Result
```

```
(define (run p)
  ;; accumulator env: Environment
  ;; invariant: Contains in-scope variable + result pairs
  (define (run/env p env)
    (match p

            …
                 ))
 (run/env p  ???  ))
```

Environments

```
;; An Environment is one of:
;; - empty
;; - (cons (list Var Result) Environment)
```

```
;; interpretation: a runtime environment
;; gives meaning to cs450lang variables
```

```
;; for duplicates, vars at front of
;; list shadow those in back
```

Think about examples where this happens!

- Needed operations:
  - `env-add    : Env Var Result -> Env`
  - `env-lookup : Env Var -> Result`

run, with an Environment accumulator

```
;; run: AST -> Result
```

```
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
     [(num n) n]
     [(add x y) (450+ (run/env x) (run/env y))]))
 (run/env p  ???  ))
```

run, with an Environment accumulator

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; run: AST -> Result
```

```
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) … (env-add env x (run/env e env)) …]
      …   ))
 (run/env p  ???  ))
```

Environment has **Results** (not **AST**)

How to convert **AST** to **Result**?

(From template!)

Be careful with "**scoping**"
(x not visible in expression e,
so use unmodified input **env**)

run, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) ??? (env-add env x (run/env e env)) …]
      …    ))
 (run/env p  ???  ))
```

run, with an Environment accumulator

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …    ))
 (run/env p  ???  ))
```

(From template!)

run body with new env containing **x**

Initial Environment?

TODO:

~~- When are variables "added" to environment~~

- What is initial environment? `empty` (for now)

```
;; run: AST -> Result
(define (run p)
  ;; accumulator env : Environment
  ;; invariant: contains in-scope var + results
  (define (run/env p env)
    (match p
      …
      [(vari x) (env-lookup env x)]
      [(bind x e body) (run/env body (env-add env x (run/env e env)))]
      …    ))
 (run/env p  ???   ))
```

`empty` ???

(for now)

Initial Environment

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (list ‘+ Expr Expr)
;; - (list ‘- Expr Expr)
```

These don’t need to be separate constructs

Put these into “initial” environment

Initial Environment

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
~~;; - (list ‘+ Expr Expr)~~
~~;; - (list ‘- Expr Expr)~~
```

Put these into “initial” environment

```
;; An Environment (Env) is one of:
;; - empty
;; - (cons (list Var Result) Env)
```

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - (Racket) Function
```

```
(define INIT-ENV
`((+ ,450+)
  (- ,450-)))
```

+ variable

Maps to our “450+” function

Initial Environment

How do users call these functions???

```
(define INIT-ENV `((+ ,450+) (- ,450-)))
```

```
(define (run p)

  ;; accumulator env : Environment
  (define (run/e p env)
    (match p
      …
     [(vari x) (lookup env x)]
     [(bind x e body) (run/e body (env-add env x (run/e e env)))]
      … ))
 (run/e p INIT-ENV    ))
```

Function Application in CS450 Lang

(initial design)

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (list ‘fncall Expr . List<Expr>)
```

function

arguments

“rest” arg

Specifies arbitrary number of args

Function Application in CS450 Lang: Examples

(initial design)

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (list ‘fncall Expr . List<Expr>)
```

function

arguments

```
(fncall + 1 2)
```

Programmers shouldn’t need to write the explicit “fncall”

Function Application in CS450 Lang: Examples

(better design)

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (cons Expr List<Expr>)
```

```
(+ 1 2)
```

Function call case (must be last, why?)

No longer need “rest” arg (why?)

Must be careful when parsing this (HW 11!)

Function Application in CS450 Lang

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (cons Expr List<Expr>)
```

parse

```
;; An AST is one of:
;; - …
;; - (vari Symbol)
;; - (bind Symbol AST AST)
;; - (call AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct call [fn args])
```

“Running” Function Calls

TEMPLATE: extract pieces of compound data

```
;; run: AST -> Result
(define (run p)


  (define (run/e p env)
    (match p
        …
      [(call fn args) (apply
                        (run/e fn env)
                        (map (curryr run/e env) args))]

        …
      ))
 (run/e p INIT-ENV))
```

```
;; An AST is one of:
;; - …
;; - (vari Symbol)
;; - (bind Symbol AST AST)
;; - (call AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct call [fn args])
```

"Running" Function Calls

```
;; An AST is one of:
;; - …
;; - (vari Symbol)
;; - (bind Symbol AST AST)
;; - (call AST List<AST>)
```

```
;; run: AST -> Result
(define (run p)

  (define (run/e p env)
    (match p
        …
      [(call fn args) (apply
                         (run/e fn env)
                         (map (curry ??? run/e env) args))]
        …
     ))
 (run/e p INIT-ENV))
```

2 arguments, can't map directly

TEMPLATE: recursive calls

"run" args before calling function – "**call by value**"

“Running” Function Calls (Function Application)

How do we actually run the function?

```
;; A Result is one of:
;; - Number
;; - UNDEFINED-ERROR
;; - (Racket) Function
```

```
;; run: AST -> Result
(define (run p)



  (define (run/e p env)
    (match p
         …
      [(call fn args) (apply
                        (run/e fn env)
                        (map (curryr run/e env) args))]

         …
      ))
 (run/e p INIT-ENV))
```

???

Applies the given function to the given args

function

List of args

(this “works” for now)

Function Application in CS450 Lang

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (cons Expr List<Expr>)
```

Function call case (must be last)

This doesn’t let users define their own functions!

Next Feature: **Lambdas?**

In-class 11/20: Write Examples

```
;; A 450LangExpr (Expr) is one of:
;; - Atom
;; - Variable
;; - ‘(bind [Var Expr] Expr)
;; - (cons Expr List<Expr>)
```

CS450LANG

```
(bind [x 10] (+ x 1))
```

Equivalent to …

RACKET

```
(let ([x 10]) (+ x 1))
```

- Repo: **cs450f24/in-class-11-20**
- File: **hw11-examples**-<*Last*>-<*First*>.rkt

Var binding

Var reference

```
(check-equal?
  (eval450 ‘(bind [x 10] x))
  10 ) ; no shadow
```

```
(check-equal?
  (eval450 ‘(bind [x 10]
              (bind [x 20]
                x)))
  20 ) ; shadow
```

```
(check-equal?
  (eval450
    ‘(bind [x 10]
        (+ (bind [x 20] x)
           x)))
  30 )
```

```
(check-equal?
  (eval450
   ‘(bind [x 10]
      (bind [x (+ x 20)]
        x)))
   30 )
```