CMSC 430 - 17 June 2026

Compiling function definitions and calls

- Iniquity - Function definitions and calls
  - designing a calling convention for Iniquity
  - arity checking at run-time

CMSC 430 - 17 June 2026

Announcements

- Exam 1 grades; regrade request due by Monday midnight
- Assignment 4 grades released later today
- Assignment 6 released later today
- Quiz released after class, due by start of class tomorrow
- Reminder: Friday is a holiday

Iniquity

Syntactic changes

Programs are now sequences of definitions and an expression

Concrete syntax:

Programs:

```
(define (f1 x11 ...) e1)
(define (f2 x12 ...) e2)
...
e
```

Expressions:

```
(f e1 ...)
```

Abstract syntax:

```
;; type Prog = (Prog (Listof Defn) Expr)
(struct Prog (ds e) #:prefab)

;; type Defn = (Defn Id (Listof Id) Expr)
(struct Defn (f xs e) #:prefab)

;; type Expr = ...
;;           | (App Id (Listof Expr))
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Prog -> Answer
(define (interp p) ...)

;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ...)
```

New kinds of error

Applying a function to the wrong number of arguments

```
#lang racket
(define (f x y) x)
(f 1)
```

We will consider this a run-time error even though we can detect this at compile time.

Future languages won't be able to statically determine arity errors.

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; ClosedExpr -> Answer
;; Prog -> Answer
(define (interp p)
  (with-handlers ([err? identity])
    (match p
      [(Prog ds e)
       (interp-e e '() ds)])))
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ;; where r closes e
  (match e
    ;...
    [(App f es)
     (let ((vs (interp-e* es r ds)))
        (match (defns-lookup ds f)
          [(Defn f xs e)
           ; check arity matches
           (if (= (length xs) (length vs))
               (interp-e e (zip xs vs) ds)
               (raise 'err))]))]))
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; (Listof Expr) Env Defns
;;     -> (Listof Value) { raises 'err }
(define (interp-e* es r ds)
  (match es
    ['() '()]
    [(cons e es)
     (cons (interp-e e r ds)
           (interp-e* es r ds))]))
```

Semantic changes

Expressions are interpreted with respect to set of definitions

```
;; Defns Symbol -> Defn
(define (defns-lookup ds f)
  (match ds
    [(cons (Defn g xs e) ds)
     (if (eq? f g)
         (Defn g xs e)
         (defns-lookup ds f))]))
```

Designing our own calling convention

A first attempt (doesn't work)

```
(f 3 4)
```

```
(define (f x y)
  (+ x y))
```

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

(Push 3)

(Push 4)

(Call ‘f)

(Pop)

(Pop)

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Ret)

Designing our own calling convention

Same thing without Call (still doesn't work)

```
(f 3 4)
```

(Push 3)

(Push 4)

(Lea ‘rax ‘r)

(Push ‘rax)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))
(Ret)

Idea: arguments passed on the stack,

return point after arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (still doesn't work)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

(Pop)

(Pop)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))
(Ret)

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes and pops

Designing our own calling convention

Return point before arguments (works!)

```
(f 3 4)
```

(Lea ‘rax ‘r)

(Push ‘rax)

(Push 3)

(Push 4)

(Jmp ‘f)

(Label ‘r)

```
(define (f x y)
  (+ x y))
```

(Label ‘f)

(compile-e (parse ‘(+ x y)) ‘(y x))

(Pop)

(Pop)

(Ret)

Idea: arguments passed on the stack,

return point *before* arguments,

caller pushes, *callee pops*

For next time

- Read Jig notes
- Take ELMS quiz