Announcements 04/07/2026

- Assignment 6: Due tonight
- Assignment 7: Due on 04/14
- Exam 2: 04/16
  - Same format as Exam 1
  - Take–home
- Today
  - Iniquity

CMSC 430: Introduction to Compilers

Iniquity: function definitions and calls

Iniquity

- Iniquity adds recursive functions, which will allow us to compute over arbitrarily large data with finite-sized programs.

```
(define (len xs)
  (if (empty? xs)
    0
    (add1 (len (cdr xs)))))

(len (cons 10 (cons 20 (cons 30 '()))))
```

AST

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

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


(struct App (f es))
```

match and

- If the first expr does not match, the entire and expr does not match.
- Otherwise, the result is the same as an and expression with the remaining exprs

```
(match (read-byte)
  [(and 99 x) (add1 x)]
  [_ #f]


(match 100
    [(and x (and 100 y)) (add1 y)])
```

apply

- **`apply proc list`** applies proc using the content of list as the (by-position) arguments.

```
(define (f x y) (+ x y)
(apply f `(1 2)
 3

(apply + '(1 2 3))
 6
```

Parser

```
(parse '(define (f x) x)
         '(f 10))

'#s(Prog (#s(Defn f (x) #s(Var x)))
 #s(App f (#s(Lit 10))))
```

Parser

- Can have multiple definitions

```
(parse '(define (f x) x)
       '(define (g x y) (+ x y))
       '(f 10))
```

- Definitions can be empty

```
(parse '(+ 1 2))

'#s(Prog () #s(Prim2 + #s(Lit 1) #s(Lit 2)))
```

Designing our own calling convention

- **Function calls are like "`let` at a distance"**

```
(f 3 4)    (define (f x y)
             (+ x y))
```

is like

```
(let ((x 3) (y 4))
  (+ x y))
```

Designing our own calling convention

- **A first attempt (doesn't work)**

```
(f 3 4)
```

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

(Push 3)

(Push 4)

(Call 'f)

(Pop)

(Pop)

(Label 'f)

(compile-e (parse '(+ x y)) '(y x))

(Ret)

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)

Designing our own calling convention

- **Return point before arguments (works!)**

```
(f 3 4)
```

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

(Lea ‘rax ‘r)
(Push ‘rax)
(Push 3)
(Push 4)
(Jmp ‘f)
(Label ‘r)

(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*

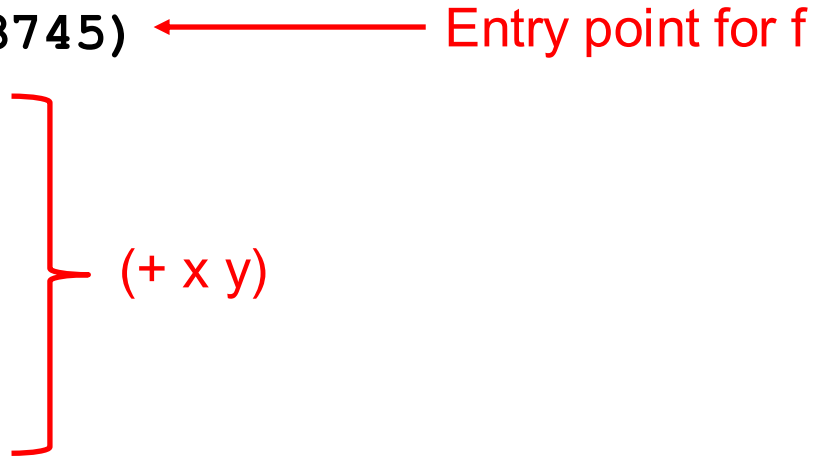
compile-define

```
;; Defn -> Asm
(define (compile-define d)
(match d
 [(Defn f xs e)
  (seq (Label (symbol->label f))
  (compile-e e (reverse xs))
  (Add rsp (* 8 (length xs))) ;pop args
(Ret))]))
```

compile-define Example

```
(compile-define
 (parse-define '(define (f x y) (+ x y)))))
```

```
(Label 'label_f_5e96933745)
(Mov 'rax (Mem 'rsp 8))
(Push 'rax)
(Mov 'rax (Mem 'rsp 8))
(Pop 'r8)
(Add 'rax 'r8)
(Add 'rsp 16)
(Ret)
```

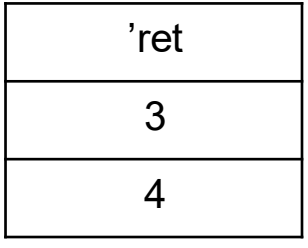


compile-app

- **The return address is placed above the arguments, so callee pops**

```
(define (compile-app f es c)
  (seq (Lea rax 'ret)
  (Push rax)
  (compile-es es (cons #f c))
  (Jmp (symbol->label f))
  (Label 'ret))))
```

`(f 3 4)`



compile-app Example

- `(compile-e (parse-e '(f 3 4)) '())`

```
(Lea 'rax (Mem 'ret))
(Push 'rax)
(Mov 'rax 48)
(Push 'rax)
(Mov 'rax 64)
(Push 'rax)
(Jmp 'label_f)
(Label 'ret)
```

`(f 3 4)`