CMSC 430 - 15 Oct 2026

Strings; function definitions and calls

- Hoax
  - Strings
  - Static memory
- Iniquity - Function definitions and calls

CMSC 430 - 15 Oct 2026

Announcements

- Assignment 5 extension: due Tuesday
- Assignment 6, 7 released (takes us through Exam 2)
- Quiz released after class, due by start of class Tuesday

Iniquity

Example Iniquity program

Concrete example

```
#lang racket
(define (add1-list lon)
  (if (empty? lon)
      '()
      (cons (add1 (car lon))
            (add1-list (cdr lon)))))

(add1-list (cons 1 (cons 2 (cons 3 '()))))
```

Example Iniquity program

Concrete example

```
#lang racket
(define (read-bytes)
  (let ((b (read-byte)))
    (if (eof-object? b)
        '()
        (cons b (read-bytes)))))

(read-bytes)
```

Example Iniquity program

Concrete example

```
#lang racket
;; copy first n bytes from input to output
(define (copy-bytes n)
  (if (zero? n)
      (void)
      (let ((b (read-byte)))
        (if (eof-object? b)
            (void)
            (begin (write-byte b)
                   (copy-bytes (sub1 n)))))))

(copy-bytes 10)
```

Example Iniquity program

Concrete example

```
#lang racket
(define (even? n)
  (if (zero? n)
      #t
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      #f
      (even? (sub1 n))))

(odd? 4097)
```

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))
```

Parsing changes

Programs are now sequences of definitions and an expression

```
;; S-Expr ... -> Prog
(define (parse . s) ...)

;; S-Expr -> Expr
(define (parse-e s) ...)

;; S-Expr -> Defn
(define (parse-define s) ...)
```

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) ...)
```

Function call example

What should this evaluate to?

Concretely:

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

```
(interp-env (App 'f (list (Lit 1) (Lit 2)))
            '()
            (list (Defn 'f '(x y)
                        (Prim2 '+ (Var 'x) (Var 'y)))))
```

Same thing as:

```
(interp-env (Prim2 '+ (Var 'x) (Var 'y))
            '((x 1) (y 2))
            (list (Defn 'f '(x y)
                        (Prim2 '+ (Var 'x) (Var 'y)))))
```

Function calls in general

What should this evaluate to?

The meaning of a function application is the meaning of the function definition's body expression, where each parameter is bound the value of the argument

```
(interp-env (App 'f (list e1 e2 ...)) r ds)
;;   where (Defn 'f (list x1 x2 ...) e) in ds
```

Same thing as:

```
(interp-env e (list (list x1 (interp-env e1 r ds))
                    (list x2 (interp-env e2 r ds))
                    ...)
              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

```
;; 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

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))
```

Except the code for f is not part of the application expression

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*