CMSC 430 - 16 June 2026

Array data; function definitions and calls

- Hoax - Vectors and strings
  - Review of make-vector
  - Dealing with the empty vector
  - Strings
  - Static memory
- Iniquity - Function definitions and calls

CMSC 430 - 16 June 2026

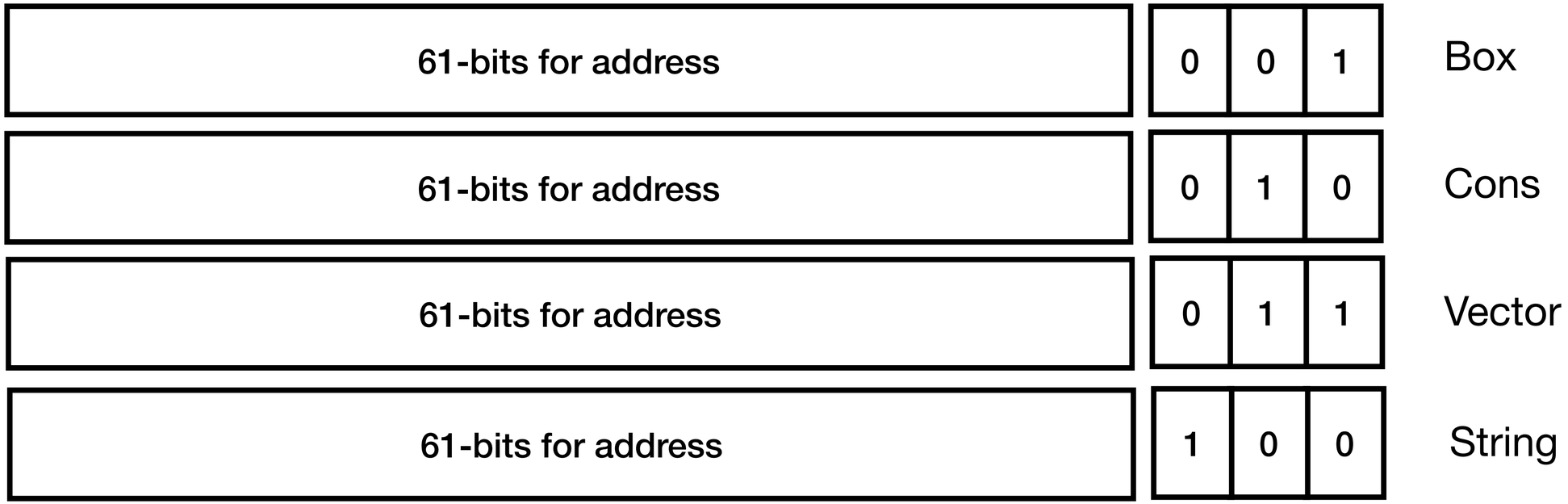
Announcements

- Exam 1 graded, released after class
- Regrade process will be posted to Piazza today
- Assignment 3 grades released
- Assignment 4 grades released later today
- `raco pkg update a86`
- Quiz released after class, due by start of class tomorrow
- Reminder: Friday is a holiday

Hoax

Encoding pointer values (Hoax)

Type tag in least significant bits



make-vector

Special case for size 0

```
(seq (Pop r8)
     (assert-natural r8)

     ; special case for length = 0
     (Cmp r8 0)
     (Jne nonzero)
     ; return canonical representation
     (Lea rax (Mem 'empty type-vect))
     (Jmp theend)
```

make-vector

When size > 0

```
; Code for nonzero case
(Label nonzero)
(Mov (Mem rbx 0) r8) ; write length
(Sar r8 1)           ; convert to bytes
(Mov r9 r8)          ; save for heap adjustment

; start initialization
(Label loop)
(Mov (Mem rbx r8) rax)
(Sub r8 8)
(Cmp r8 0)
(Jne loop)
; end initialization
```

make-vector

Constructing return value and account for memory use

```
(Mov rax rbx)
(Xor rax type-vect)  ; create tagged pointer
(Add rbx r9)         ; acct for elements and stored length
(Add rbx 8)
(Label theend)))]
```

Static memory

Code lives in memory, so you can make data part of the program

```
; create '#(#t #f 99)
(Mov rax (value->bits 3))
(Mov (Mem rbx 0) rax)
(Mov rax (value->bits #t))
(Mov (Mem rbx 8) rax)
(Mov rax (value->bits #f))
(Mov (Mem rbx 16) rax)
(Mov rax (value->bits 99))
(Mov (Mem rbx 24) rax)
(Mov rax rbx)
(Xor rax type-vect)
(Add rbx 32)
```

If you know the vector you want to make at compile-time, you don't need to do this…

Static memory

Code lives in memory, so you can make data part of the program

```
(Data)
; create '#(#t #f 99)
(Label 'myvect)
(Dq (value->bits 3))
(Dq (value->bits #t))
(Dq (value->bits #f))
(Dq (value->bits 99)))
```

…you can statically allocate memory in the data section.

And create a tagged pointer to it.

```
(Lea rax (Mem 'myvect type-vect))
```

Static memory

Code lives in memory, so you can make data part of the program

```
(Data)
; create '#()
(Label 'empty)
(Dq (value->bits 0))
```

Very useful for canonically representing the empty vector.

make-vector

Special case for size 0

```
(seq (Pop r8)
     (assert-natural r8)

     ; special case for length = 0
     (Cmp r8 0)
     (Jne nonzero)
     ; return canonical representation
     (Lea rax (Mem 'empty type-vect))
     (Jmp theend)
```

Hoax: Adding strings

Concrete examples

```
(make-string 3 #\t)                    ;=> "ttt"
(let ((s (make-string 3 #\t)))
  (string-ref s 0))                    ;=> #\t
(let ((s (make-string 3 #\t)))
  (string-ref s 3))                    ;=> err
(let ((s (make-string 3 #\t)))
  (string-length s))                   ;=> 3
"fred"                                 ;=> "fred"
```

Hoax: Adding strings

AST additions

```
;; type Lit = ... | String
;; type Op1 = ...
;;          | 'string? | 'string-length
;; type Op2 = ...
;;          | 'make-string | 'string-ref
```

Hoax: Adding strings

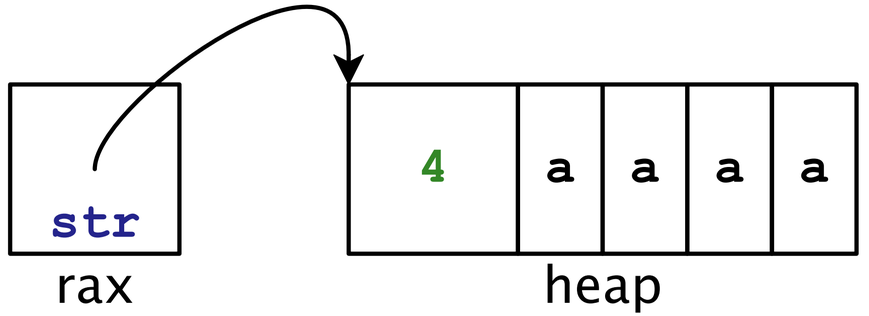
Semantic additions

```
;; type Value = ...
;; | (string Char ...)
```

Creating a string

A string is a tagged pointer to a sized array of *codepoints* in memory

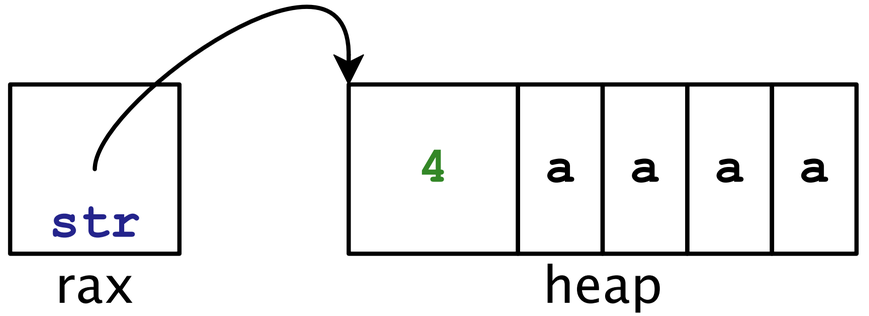
```
> (exec (parse '(make-string 4 #\a)))
"aaaa"
```



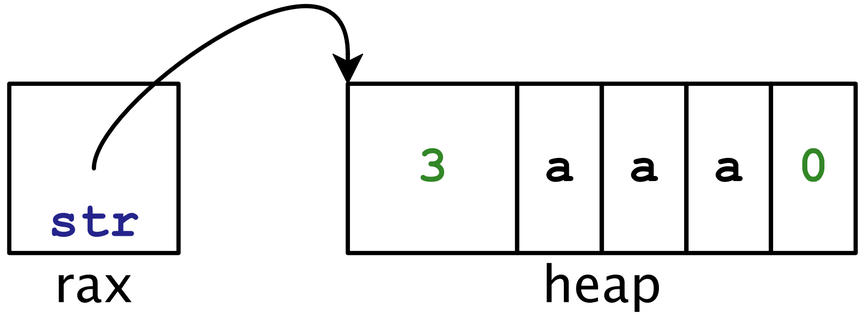
Creating a string

A string is a tagged pointer to a sized array of *codepoints* in memory

```
> (exec (parse '(make-string 4 #\a)))
"aaaa"
```




```
> (exec (parse '(make-string 3 #\a)))
"aaa"
```



string-length

Just like vector-length

```
(seq (assert-string rax)
     (Mov rax (Mem rax (- type-str))))
```

string-ref

Similar to vector-ref, but different

```
(seq (Pop r8)
     (assert-natural rax)
     (assert-string r8)
     (Mov r9 (Mem r8 (- type-str)))
     (Cmp rax r9)
     (Jge 'err)
     (Sar rax 2)
     (Mov eax (Mem r8 rax (- 8 type-str)))
     (Sal rax char-shift)
     (Xor rax type-char))
```

string-ref

```
(seq (Pop r8)
     (assert-natural rax)
     (assert-vector r8)
     (Mov r9 (Mem r8 (- type-vect)))
     (Cmp rax r9)
     (Jge 'err)
     (Sar rax 1)
     (Mov rax (Mem r8 rax (- 8 type-vect))))
```

vector-ref

string-set!

We don't include, thus strings are immutable

[This space intentionally blank.]

make-string

Similar to make-vector, but different

```
(seq (Pop r8)
     (assert-natural r8)

     ; special case for length = 0
     (Cmp r8 0)
     (Jne nonzero)
     ; return canonical representation
     (Lea rax (Mem 'empty type-str))
     (Jmp theend)

     ; Code for nonzero case
     (Label nonzero)
     (Mov (Mem rbx 0) r8) ; write length
     (Sar r8 2)           ; convert to bytes
     (Mov r9 r8)          ; save for heap adjustment

     (Sar rax char-shift) ; convert to codepoint

     ; start initialization
     (Label loop)
     (Mov (Mem rbx r8 4) eax)
     (Sub r8 4)
     (Cmp r8 0)
     (Jne loop)
     ; end initialization

     (Mov rax rbx)
     (Xor rax type-str)   ; create tagged pointer
     (Add rbx r9)         ; acct for elements and stored length
     (Add rbx 8)
     ; Pad to 8-byte alignment
     (Add rbx 4)
     (Sar rbx 3)
     (Sal rbx 3)
     (Label theend)))
```

Let's take it in pieces

make-string (1/8)

Similar to make-vector, but different

```
(Pop r8)
(assert-natural r8)
```

Just like `make-vector`

make-string (2/8)

Similar to make-vector, but different

```
(assert-char rax)
```

Unlike `make-vector`

make-string (3/8)

Similar to make-vector, but different

```
; special case for length = 0
(Cmp r8 0)
(Jne nonzero)
; return canonical representation
(Lea rax (Mem 'empty type-str))
(Jmp theend)
```

Just like `make-vector`, but makes an empty *string*

make-string (4/8)

Similar to make-vector, but different

```
; Code for nonzero case
(Label nonzero)
(Mov (Mem rbx 0) r8) ; write length
(Sar r8 2)           ; convert to bytes
(Mov r9 r8)          ; save for heap adjustment
```

Just like `make-vector`, but half as many bytes

make-string (5/8)

Similar to make-vector, but different

```
(Sar rax char-shift) ; convert to codepoint
```

Unlike `make-vector`, don't write value, write codepoint

make-string (6/8)

Similar to make-vector, but different

```
; start initialization
(Label loop)
(Mov (Mem rbx r8 4) eax)
(Sub r8 4)
(Cmp r8 0)
(Jne loop)
; end initialization
```

Just like `make-vector`, but writing 4 bytes at a time

make-string (7/8)

Similar to make-vector, but different

```
(Mov rax rbx)
(Xor rax type-str)   ; create tagged pointer
(Add rbx r9)         ; acct for elements and stored length
(Add rbx 8)
```

Just like `make-vector`

make-string (8/8)

Similar to make-vector, but different

```
; Pad to 8-byte alignment
(Add rbx 4)
(Sar rbx 3)
(Sal rbx 3)
```

Unlike `make-vector`, need to pad to preserve heap invariant when `string-length` is odd

String literals

String literals are compiled to static memory

```
> (compile-e (Lit "wilma") '())
(list
 (Data)
 (Label 'string435421)
 (Dq 80)
 (Dd 119)
 (Dd 105)
 (Dd 108)
 (Dd 109)
 (Dd 97)
 (Dd 0)
 (Text)
 (Lea 'rax (Mem 'string435421 4)))
```

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)
```

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

For next time

- Read Iniquity notes
- Take ELMS quiz