CMSC 430 - 8 Oct 2026

Array data

- Hoax - Vectors and strings

CMSC 430 - 8 Oct 2026

Announcements

- Exam 1 grades posted
- Assignment 4 due tonight, midnight
- Assignment 5 released; due Thursday
- New quiz out, due Thursday before class
- Reminder: Tuesday is a holiday

Hints for Assignment 5

Some thoughts on the assignment

- Determining if something is a list requires traversing it
- Computing the length of a list requires traversing it
- Temporal and spatial ordering of memory reads/writes can be arbitrary
  - You can write the elements starting from the end
  - You can write the length last, etc.
- Working examples out on paper will get you there faster
  - Draw the memory you expect
- Using static data to make examples that you interactively develop can help
- All of the primitives can be implemented in a single pass over input and without allocating temporary memory (i.e. only allocating the output value)
- Reference solution < 90 LOC
- Programming in assembly sucks, someone should develop a high-level… oh

Hoax

Hoax: Adding vectors

Concrete examples

```
(make-vector 3 #t)                  ;=> '#(#t #t #t)
(let ((v (make-vector 3 #t)))
  (vector-ref v 0))                 ;=> #t
(let ((v (make-vector 3 #t)))
  (vector-ref v 3))                 ;=> err
(let ((v (make-vector 3 #t)))
  (vector-length v))                ;=> 3
(let ((v (make-vector 3 #t)))
  (begin (vector-set! v 0 #f)
         v))                        ;=> '#(#f #t #t)
```

Hoax: Adding vectors

AST additions

```
;; type Op1 = ...
;;          | 'vector? | 'vector-length
;; type Op2 = ...
;;          | 'make-vector | 'vector-ref
;; type Op3 = 'vector-set!
```

Hoax: Adding vectors

Semantic additions

```
;; type Value = ...
;; | (vector Value ...)
```

Hoax: Adding vectors

Semantic additions

```
;; Op2 Value Value -> Value { raises 'err }
(define (interp-prim2 op v1 v2)
  (match (list op v1 v2)
    #;...
    [(list 'make-vector (? integer?) _)
     (if (<= 0 v1)
         (make-vector v1 v2)
         (raise 'err))]
    [(list 'vector-ref (? vector?) (? integer?))
     (if (<= 0 v2 (sub1 (vector-length v1)))
         (vector-ref v1 v2)
         (raise 'err))]))
```

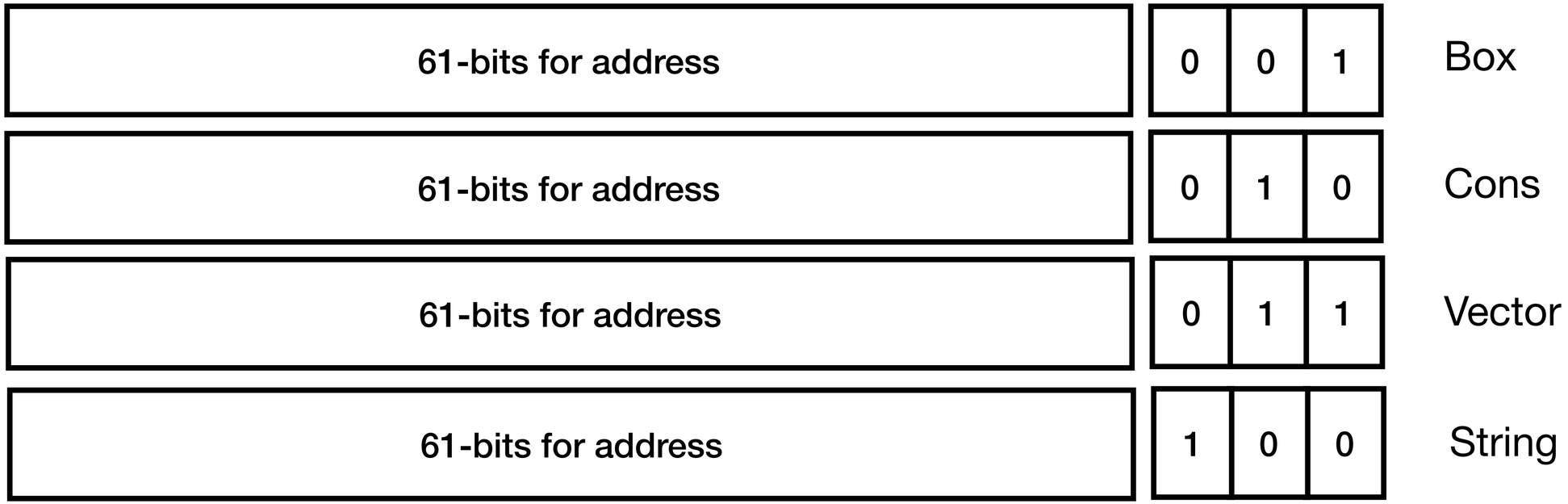
Hoax: Adding vectors

Semantic additions

```
;; Op3 Value Value Value -> Value { raises 'err }
(define (interp-prim3 p v1 v2 v3)
  (match (list p v1 v2 v3)
    [(list 'vector-set! (? vector?) (? integer?) _)
     (if (<= 0 v2 (sub1 (vector-length v1)))
         (vector-set! v1 v2 v3)
         (raise 'err))]
    [_ (raise 'err)]))
```

Encoding pointer values (Hoax)

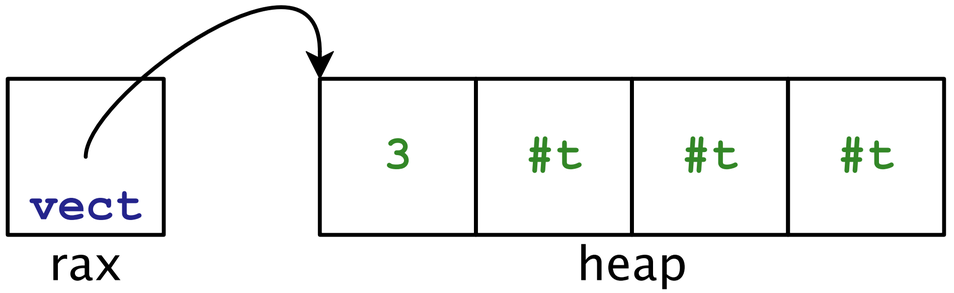
Type tag in least significant bits



Creating a vector

A vector is a tagged pointer to a sized array of values in memory

```
> (exec (parse '(make-vector 3 #t)))
'#(#t #t #t)
```



Making an example vector

Write 4 words of memory

```
; 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)
```

Accessing parts of a vector

Assume rax holds a vector, index is in bounds

```
;; access vector-length
(Mov rax (Mem rax (- type-vect)))

;; access first element (index 0)
(Mov rax (Mem rax (- 8 type-vect)))

;; access second element (index 1)
(Mov rax (Mem rax (- 16 type-vect)))

;; access third element (index 2)
(Mov rax (Mem rax (- 24 type-vect)))
```

Accessing parts of a vector

Check rax holds a vector, assume index is in bounds

```
;; access vector-length
(assert-vector rax)
(Mov rax (Mem rax (- type-vect)))

;; access first element (index 0)
(assert-vector rax)
(Mov rax (Mem rax (- 8 type-vect)))

;; ...
```

```
> (assert-vector 'rax)
(list
 (Push 'rax)
 (And 'rax 7)
 (Cmp 'rax 3)
 (Pop 'rax)
 (Jne 'err))
```

Implementing vector-length primitive

Vector is in rax

```
['vector-length
 (seq (assert-vector rax)
      (Mov rax (Mem rax (- type-vect))))]
```

Implementing vector-ref primitive

Vector is on stack, index is in rax

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

Implementing vector-set! primitive

Vector is on stack, index is next on stack, new element in rax

```
['vector-set!
 (seq (Pop r10)
      (Pop r8)
      (assert-vector r8)
      (assert-natural r10)
      (Mov r9 (Mem r8 (- type-vect)))
      (Cmp r10 r9)
      (Jge 'err)
      (Sar r10 1) ; convert to byte offset
      (Mov (Mem r8 r10 (- 8 type-vect)) rax)
      (Mov rax (value->bits (void))))]
```

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.

Implementing make-vector primitive

Size is on stack, initialization element in rax

```
['make-vector
 (let ((nonzero (gensym 'nz))
       (loop (gensym 'loop))
       (theend (gensym 'theend)))

   (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)

        ; 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

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

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)))]
```

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

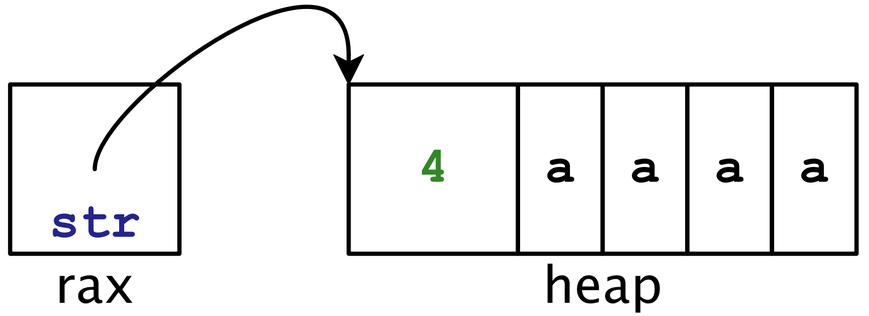
Semantic additions

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

Creating a string

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

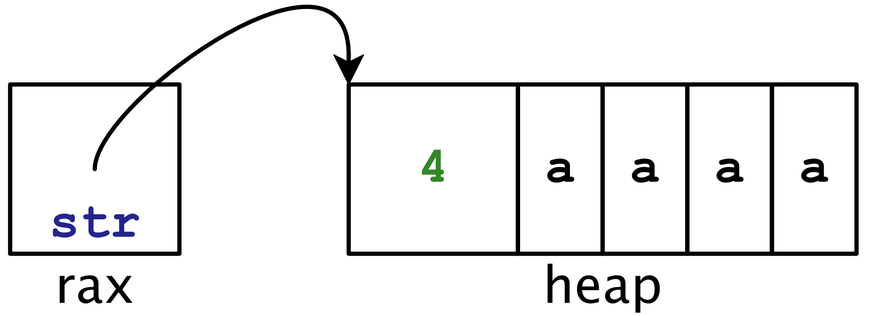
```
> (exec (parse '(make-string 4 #\a)))
"aaaa"
```



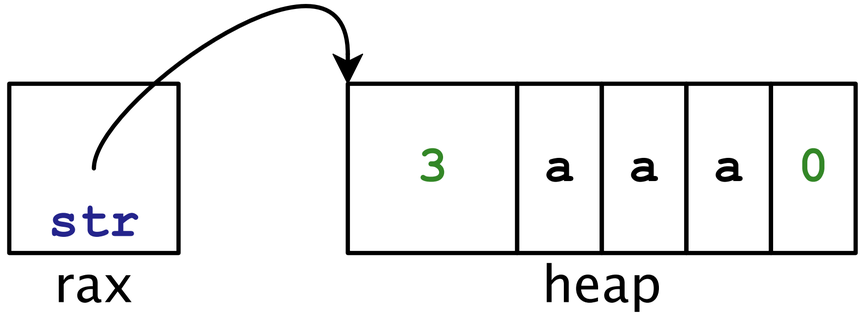
Creating a string

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)))
```