



CSE 231: Heap Allocated Data

Ranjit Jhala May 5, 2026

Big Data!

63-bit →

The stack suffices for *little* data like `int`, `bool` whose size known at compile time, but what of *unbounded* structures like lists, trees, graphs?

(Two-element) Arrays

`(vec e1 e2)`
`(vec 0 (vec 1 (vec 2 (vec 3 nil))))`

Lets us write *structs*, *lists*, ...

```
<expr> := ... | nil | (vec <expr> <expr>)  
          | (vec-get <expr> 0) | (vec-get <expr> 1)
```

`(vec <expr>+)`
`(vec-get <expr> <expr>)`

- `nil` is the *empty* vector
- `(vec e1 e2)` constructs a vector of *two* elements
- `(vec-get e i)` returns the *i*th element of vector *e*

Helpers: Head and Tail

Lets define some *helpers* to work with `vec`

```
(fun (head v) (vec-get v 0)) ; get first element  
(fun (tail v) (vec-get v 1)) ; get second element
```

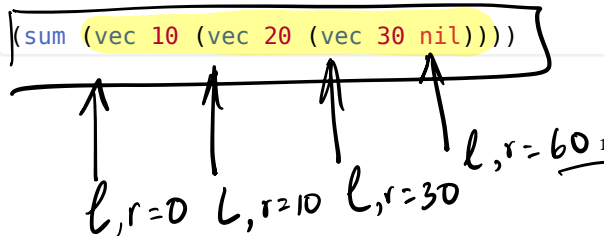
`e 0 = nil`
`(vec e1 e2)`
`(vec-get e 0)`
`(vec-get e 1)`

QUIZ: Semantics

What should the following programs *evaluate* to

Program	Result
<code>nil</code>	<code>nil</code>
<code>(vec (+ 5 5) (+ 10 10))</code>	<code>(vec 10 20)</code>
<code>(let (p (vec 10 20)) (+ (head p) (tail p)))</code>	<code>30</code>
<code>(fun (sum l) (let (r 0) (loop (if (= l nil) (break r) (block (set! r (+ r (head l))) (set! l (tail l)))))))</code>	<code>60</code>
<code>(sum (vec 10 (vec 20 (vec 30 nil))))</code>	

10 20
vector located at 0x....
(10, 20)



QUIZ: Abstract Syntax

Lets write the *abstract syntax* for our language

```
enum Expr { // ...
}

```

```
enum Index {
}

```

QUIZ: Evaluator Value

How to extend Val and eval to support vec?

```
enum Val {
  Num(i32),
  Bool(bool),
  MyVec(Vec<Val>)
}

```

```
enum Expr {
  Num(i32),
  Bool(bool),
  ...
}

```

```
fn eval(e, env, funs) -> Val {
  match e {
    Expr::Nil => {
      MyVal (vec![])
    }
    Expr::Vec(e1, e2) => {
      let vi = eval(ei, ...)?
      MyVal (vec![v1, v2])
      _____
      _____
      _____
    }
    Expr::Get(e, i) => {
      _____
      _____
      _____
    }
  }
}

```

23, 59
true, false

QUIZ: Where to store the vectors?

Why can't we just store them on the stack?

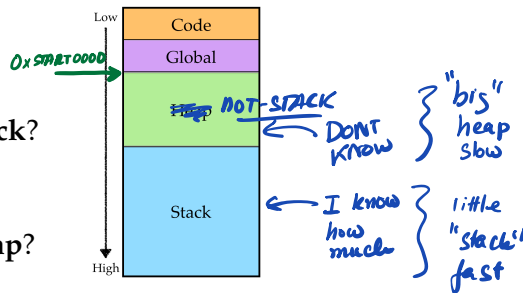
- ① how much spec? what about GROW / SHRINK
 - ② how to PASS as a param? How MUCH SPACE!
 - ③ where do locals LIVE on stack
- (let (x (if cond
(vec-of-size 2)
(vec-of-size 200)))



Stack vs Heap

(y ...)

Why do we need a stack and heap?



What would happen without the stack?

(+ true 10)

What would happen without the heap?

(let (p (vec 1 2))
(+ p 10))

Compiling Vectors

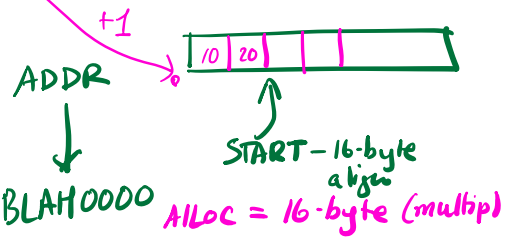


This program defines and uses a let-bound vector p

```
(let [p (vec 10 20)]  
  (+ (vec-get p 0) (vec-get p 1)))
```

1. What stack slot does p live in?
2. What value should we put in that slot?

ENFORCE last 4 BITS of ADDR = 0?



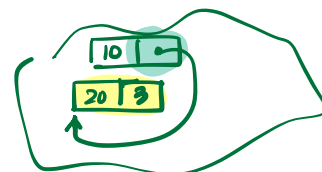
Representing Vectors

Value	Representation
number	xxx0
false (3)	x011
true (7)	x111
nil (9)	1001
(vec ...)	0001

num 0 1 bool 0 1
vec nil

(vec 10 (vec 20 3))

QUIZ: How to ensure x001 is valid vec?



(vec 10 20 30 40)

Runtime: Allocate Heap

1. **Allocate** a large chunk of memory to serve as the heap

```
#[repr(align(16))]
struct AlignedHeap([u64; 100000]);
static mut HEAP = AlignedHeap([0; 100000]);
```

2. **Pass** a pointer to the heap to asm code

```
#[link_name = "\x01our_code_starts_here"]
fn our_code_starts_here(input: i64, heap: *mut u64) -> i64;

fn main() {
    // ...
    let i = unsafe {
        our_code_starts_here(input, HEAP.0.as_mut_ptr())
    };
    // ...
}
```

QUIZ: How does asm know where heap is?

Runtime: Printing Values

```
fn print_val(val: i64) {
    if val == FALSE { print!("false"); }
    else if val == TRUE { print!("true"); }
    else if val & 1 == 0 { print!("{}", val >> 1); }
    else if val == NIL { print!("nil"); }
    else { print_vec(val); }
}

fn print_vec(val: i64) {
    let ptr = (val - 1).try_into().unwrap();
    let ptr: *const i64 =
        std::ptr::with_exposed_provenance::<i64>(ptr);
    let val1 = unsafe { *ptr };
    let val2 = unsafe { *ptr.add(1) };
    print!("(vec ");
    print_val(val1);
    print!(" ");
    print_val(val2);
    print!(")");
}
```

QUIZ: Assembly for Constructor (vec)

Program

nil

Assembly

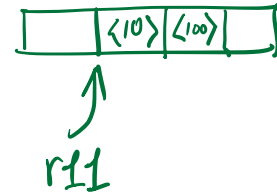
```
mov rax, {NIL}
```

(vec 10 100)

```
; r11 ≡ next avail heap addr
```

(vec e1 e2)

```
[r11] ← <e1>
[r11+8] ← <e2>
rax ← r11
r11 ← r11 + 16
rax ← rax + 1
```



QUIZ: Assembly for Accessor (vec-get)

Program

```
(let (p (vec 10 100))
  (vec-get p 0))
```

Assembly

```
; <(vec 10 100)>
mov rax, [rbp - 8·{P}]
; check type:
sub rax, 1
mov rax, [rax + 0·8]
```

```
(let (p (vec 10 100))
  (vec-get p 1))
```

```
; <(vec 10 100)>
mov rax, [rbp - 8·{P}]
; check type:
sub rax, 1
mov rax, [rax + 1·8]
```

(vec-get e idx)

```
; <e>
; check type:
sub rax, 1
mov rax, [rax + idx·8]
```

How to generalize to arbitrary-length vectors?

Syntax?

Runtime representation?