CSE 231: Heap Allocated Data

Ranjit Jhala *May 5, 2026*

Big Data!

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

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

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

- `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
```

QUIZ: Semantics

What should the following programs *evaluate* to

Program	**Result**
`nil`	
`(vec (+ 5 5) (+ 10 10))`	
`(let (p (vec 10 20)) (+ (head p) (tail p)))`	

`(fun (sum l)`
`  (let (r 0)`
`  (loop`
`  (if (= l nil)`
`  (break r)`
`  (block (set! r (+ r (head l)))`
`  (set! l (tail l)))))))`
 	
`(sum (vec 10 (vec 20 (vec 30 nil))))`	

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),



}
```

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



}
```

```
fn eval(e, env, funs) -> Val {
  match e {
    Expr::Nil => {

      ____________________________
    }
    Expr::Vec(e1, e2) => {

      ____________________________

      ____________________________

      ____________________________

    }
    Expr::Get(e, i) => {

      ____________________________

      ____________________________

      ____________________________

    }
}
```

QUIZ: Where to store *the vectors?*

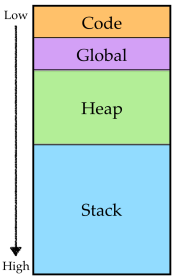
Why can't we *just* store them on the stack?

Stack vs Heap

Why do we need a **stack** *and* **heap**?

What would happen *without* the **stack**?

What would happen *without* the **heap**?



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?

Representing Vectors

Value	**Representation**
`number`	`xxx0`
`false`	`x011`
`true`	`x111`
`nil`	`1001`
`(vec ...)`	`0001`

QUIZ: How to ensure `x001` is valid `vec`?

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 the `asm` code know where *the 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	Assembly
`nil`	
`(vec 10 100)`	
`(vec e1 e2)`	

QUIZ: Assembly for Accessor (`vec-get`)

Program	Assembly
`(let (p (vec 10 100)) (vec-get p 0))`	`; <(vec 10 100)>`
`(let (p (vec 10 100)) (vec-get p 1))`	`; <(vec 10 100)>`
`(vec-get e idx)`	`; <e>`

How to generalize to arbitrary-length vectors?

Syntax?

Runtime representation?