



EECS 483: Compiler Construction

Lecture 11:

Dynamic Typing Continued, Heap Allocation

February 18

Winter Semester 2026

Learning objectives

Design space for dynamically typed values

Implementation of dynamic type checks in assembly, SSA

Stack vs Heap Allocation

Mutable Arrays syntax and semantics

Implementation of Heap-allocated, large datatypes

State of the Snake Language



Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures (pushdown automata if you don't use extern functions)

Snake v4: **Diamondback**

1. Add new datatypes, use dynamic typing to distinguish them at runtime
2. Include heap-allocated variable-sized arrays, allowing for unrestricted memory usage

Computational power: Turing complete

Booleans in Boa/Cobra

In Boa/Cobra, booleans and integers weren't truly distinct datatypes.

- All integers could be used in logical operations
- All booleans could be used in arithmetic operations

Booleans in Boa/Cobra

The following are all valid programs with well-defined semantics in Boa/Cobra

`-1 && 3`

`true + 5`

`7 >= false`

Let's change the language semantics so these are **errors** instead.

Booleans in Boa/Cobra

Can we implement operations **isInt** and **isBool** that distinguish between integers and booleans?

```
isInt(true) == false
```

```
isInt(1) == true
```

No, true and 1 have the exact same representation at runtime

Code Example: Dynamic Typing Interpreter

Let's explore what the **semantics** of dynamic typing is by looking at an **interpreter** for Boa that uses dynamic typing instead of coercions.

Semantics of Dynamic Typing

- A Snake value is not just an int anymore. It is **either** an int or a boolean, and we need to be able to tell the difference at runtime in order to determine when we should error and how to implement `isInt`, `isBool`.
- Many operations can now produce runtime errors if type tags are incorrect, need to specify
 - what the appropriate error messages are
 - evaluation order between expressions executing and type tags
 - `true + (let _ = print(3) in 3)`
 - does this print 3 before it errors?

Representing Dynamically Typed Values

In Adder/Boa/Cobra, all runtime values were integers.

In Diamondback, a runtime value must have both a type tag and a value that matches the type tag

How should we represent tags and values in our compiled program?

Representing Dynamically Typed Values

Approach 1: Values as 8 bytes, Tags as extra data

A snake value is 9 bytes

the first byte is a tag: 0x00 for integer, 0x01 for boolean. Use a full byte to keep our values byte-aligned

the remaining 64 bits are the underlying integer, bool or pointer

Upside: Faithful representation of our Rust interpreter

Downside: 1 byte memory overhead for all values plus padding, calling convention and architecture are 8-byte oriented, tedious to implement pervasively

Representing Dynamically Typed Values

Approach 2: Values as pointers

A snake value is a 64-bit pointer to an object on the **heap**

value stored on the heap can then be whatever size we want, the pointer is always 64 bits.

store a tag and value on the heap similarly to previous approach.

A value stored in this way is called **boxed**.

Representing Dynamically Typed Values

Approach 2: Values as pointers

A snake value is a 64-bit pointer to an object on the **heap**

value stored on the heap can then be whatever size we want, the pointer is always 64 bits.

store a tag and value on the heap similarly to previous approach.

Upside: uniform implementation, 64-bit values can be compiled as before

Downside: Adds memory overhead pervasively.

Representing Dynamically Typed Values

Approach 3: compromise

A snake value is a 64-bit value.

Use the least significant bits of the value as a **tag**.

Represent simple data like integers, booleans within the 64-bits

Represent large datatypes like arrays, closures, structs as pointers to the heap

Upside: use stack allocation more often

Downside: can't fit 64 bits and a tag...

Roughly the approach used in high-performance Javascript engines (v8) as well as some garbage-collected typed languages (OCaml)

Representing Dynamically Typed Values

To implement our compiler, we need to specify

1. How each of our Snake values are represented at runtime
2. How to implement the primitive operations on these representations

Integers

Implement a snake integer as a 63-bit signed integer followed by a 0 bit to indicate that the value is an integer

Number	Representation
1	0b00000000_0000.....0000_00000010
6	0b00000000_0000.....0000_00001100
-1	0b11111111_1111.....1111_11111110

I.e., represent a 63-bit integer n as the 64-bit integer $2 * n$

Booleans

The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b01` to distinguish from integers and other datatypes

Use the remaining 62 bits to encode true and false as before as 1 and 0

Number	Representation
<code>true</code>	<code>0b00000000_0000.....0000_00000101</code>
<code>false</code>	<code>0b00000000_0000.....0000_00000001</code>

$2^{62} - 2$ bit patterns are therefore "junk" in this format

Boxed Data

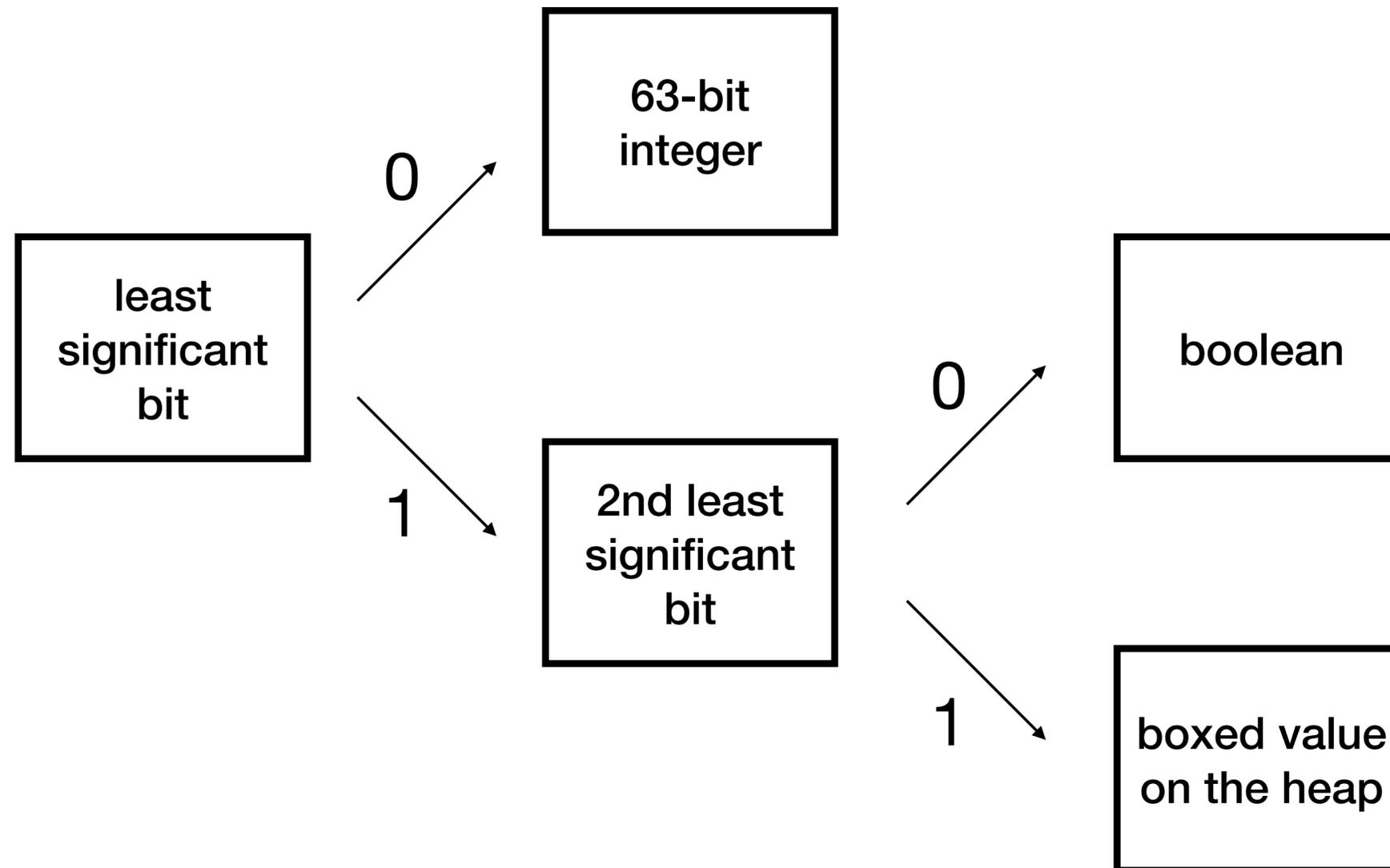
The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b11` to distinguish from booleans.

Use remaining 62-bits to encode a pointer to the data on the heap

Why is this ok? Discuss more thoroughly on Wednesday

Representing Dynamically Typed Values



Implementing Dynamically Typed Operations

We need to revisit our implementation of all primitives in assembly code to see how they should work with our new datatype representations.

1. Arithmetic operations (add, sub, mul)
2. Inequality operations (<=, <, >=, >)
3. Equality
4. Logical operations (&&, ||, !)

As well as supporting our new operations isInt and isBool

Implementing Dynamically Typed Operations

In dynamic typing, implementing a primitive operation has two parts:

1. How to check that the inputs have the correct type tag
2. How to actually perform the operation on the encoded data

Implementing Dynamically Typed Operations

Live code

Implementing Dynamically Typed Operations

We need to revisit our implementation of all primitives in assembly code to see how they should work with our new datatype representations.

1. Arithmetic operations (add, sub, mul)
2. Inequality operations (<=, <, >=, >)
3. Equality
4. Logical operations (&&, ||, !)

As well as supporting our new operations isInt and isBool

Implementing Dynamically Typed Operations

In dynamic typing, implementing a primitive operation has two parts:

1. How to check that the inputs have the correct type tag
2. How to actually perform the operation on the encoded data

Implementing Dynamically Typed Operations

Live code

Compiling Dynamic Typing

We know what the source semantics is and what kind of assembly code we want to generate.

In implementing the compiler, we now we have a design choice: in what phase of the compiler do we actually "implement" dynamic typing?

1. Implement everything in x86 code generation
2. Implement everything in lowering to SSA
3. Implement in multiple passes

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

In this case, SSA values would be dynamically typed, like Diamondback

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

Diamondback

`x + y`

SSA

`r = x + y`
`ret r`

x86

```
mov rax, [rsp - 8]
test rax, 1
jnz err_arith_exp_int
mov r10, [rsp - 8]
test r10, 1
jnz err_arith_exp_int
add rax, r10
ret
```

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

In this case, SSA values would be dynamically typed, like Diamondback

Downside: goes against the philosophy that SSA should be thin wrapper around the assembly code.

Makes the semantics of SSA more complex and so the code generation more complex.

More complex code generation: missed opportunities for SSA-based optimization

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

In this approach, SSA values are as before always 64-bit integers, and SSA operations work on these 64-bit integers (as they do now)

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

Diamondback

```
x * y
```

SSA

```
check_y():  
    y_bit = y & 1  
    c = y_bit == 0  
    cbr mult_xy() err()  
mult_xy():  
    tmp = x * y  
    z = tmp >> 1  
    ret z  
x_bit = x & 1  
b = x_bit == 0  
cbr check_y() err()
```

...

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

Benefit: code generation is very simple, at cost of SSA lowering more complex

Downside: difficult to optimize unnecessary tag checks away

Compiling Dynamic Typing

Approach 3: implement dynamic typing in multiple passes

In lowering to SSA, make some aspects of dynamic typing explicit but leave the tag checking as primitive operations.

Implement the tag checking in the x86 code generation.

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

Diamondback

`x * y`

SSA

`assertInt(x)`

`assertInt(y)`

`half = x >> 1`

`r = half * y`

`ret r`

Insert type tag assertions in SSA, implement bit-twiddling manually

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

```
SSA
assertInt(x)
```

```
x86
mov rax, [rsp - offset(x)]
test rax, 1
jnz assert_int_fail
...
assert_int_fail:
  sub rsp, 8
  call snake_assert_int_error
```

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):  
    if x == 0:  
        1  
    else:  
        x * fact(x - 1)  
  
in  
fact(7)
```

SSA

```
...  
tmp1 = x - 1  
tmp2 = call fact(tmp1)  
assertInt(x)  
assertInt(tmp2)  
r = x * tmp2  
ret r
```

will these **assertInt** ever fail?

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):  
    if x == 0:  
        1  
    else:  
        x * fact(x - 1)  
  
in  
fact(7)
```

SSA

```
...  
tmp1 = x - 1  
tmp2 = call fact(tmp1)  
assertInt(x)  
assertInt(tmp2)  
r = x * tmp2  
ret r
```

with a simple **static analysis** determine that `x`, `tmp2` always have the correct tag for an `Int`. Remove unnecessary assertions

Compiling Dynamic Typing

Compare to approach 2:

Diamondback

```
x * y
```

how would we remove the checking from the code on the right?

SSA

```
check_y():  
    y_bit = y & 1  
    c = y_bit == 0  
    cbr mult_xy() err()  
mult_xy():  
    tmp = x * y  
    z = tmp >> 1  
    ret z  
x_bit = x & 1  
b = x_bit == 0  
cbr check_y() err()
```

...

Summary: Adding Dynamic typing

How does adding dynamic typing affect each pass of our compiler?

Changes to Frontend

New error: only 63-bit integers are supported, so need to reject 64-bit values in the parser/frontend

Changes to Middle End

Diamondback values: tagged data, either a 63-bit int, or true or false

SSA values: 64-bit integers

Add primitive assertions **assertInt** and **assertBool** to SSA

Use bitwise masking and left/right shift in SSA to encode the correct semantics of Diamondback values and operations

Changes to Back End

Implement **assertInt** and **assertBool** operations in x86, calling out to functions implemented in Rust to display appropriate errors

Changes to Runtime (stub.rs)

Parse input arguments into snake values.

Update printing to account for new representation

Implement functions that display runtime errors and exit the process

State of the Snake Language



Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures, extern (pushdown automata)

Snake v4: **Diamondback**

1. Add new datatypes, use dynamic typing to distinguish them at runtime
- 2. Include heap-allocated variable-sized arrays, allowing for unrestricted memory usage**

Computational power: Turing complete

Extending the Snake Language

Diamondback: Arrays



```
def main(x):  
    [x , x + 1, x + 2]
```

allocate an array with a statically known size

Extending the Snake Language

Diamondback: Arrays

```
def main(x):  
    newArray(x)
```

allocate an array with dynamically determined size (elements initialized to 0)



Extending the Snake Language

Diamondback: Arrays



```
def main(x):  
    let a = [x , -1 * x ] in  
    a[0]
```

array indexing

Extending the Snake Language

Diamondback: Arrays



```
def main(x):  
  let a = [x, -1 * x] in  
  let _ = a[1] := a[1] + 1 in  
  a[1]
```

arrays can be mutably updated

Extending the Snake Language

Diamondback: Arrays



```
def main(x):  
    let a = [x , -1 * x ] in  
    length(a)
```

should be able to access the length of any array value

Extending the Snake Language

Diamondback: Arrays



```
def main(x):  
    let a = [x , -1 * x ] in  
    a[3]
```

Out of bounds access/update should be runtime errors

Extending the Snake Language

Diamondback: Arrays

```
def main(x):  
    let a = [x , -1 * x ] in  
        isArray(a)
```

support tag checking as with ints, bools



Extending the Snake Language

Diamondback: Arrays

```
def main(x):  
  let list = [0, 1, false] in  
  let _ = list[2] := list in  
  ...
```

mutable updates allow for cyclic data



Concrete Syntax



```
<expr>: ...  
| <array>  
| <expr> [ <expr> ]  
| <expr> [ <expr> ] := <expr>  
| newArray ( <expr> )  
| isBool ( <expr> )  
| isInt ( <expr> )  
| isArray ( <expr> )  
| length ( <expr> )
```

```
<exprs>:  
| <expr>  
| <expr> , <exprs>  
  
<array>:  
| [ ]  
| [ <exprs> ]
```

Abstract Syntax

```
enum Prim {  
    ...  
    // Unary  
    IsArray,  
    IsBool,  
    IsInt,  
    NewArray,  
    Length,  
  
    MakeArray, // 0 or more arguments  
    ArrayGet, // first arg is array, second is index  
    ArraySet, // first arg is array, second is index, third is new value  
}
```



Extending the Snake Language

Diamondback: Arrays



Semantics:

1. Each time we allocate an array should be a new memory location, so that updates don't overwrite previous allocations
2. What value does $e1[e2] := e3$ produce?
options: a constant, the value of $e1$ or $e3$, the old value of $e1[e2]$
3. Is equality of arrays by value or by reference?

$[0, 1, 2] == [0, 1, 2]$

Allocating Arrays

Where should the contents of our arrays be stored?

- Stack?
- Heap?

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):  
  let a = [x , -1 * x ] in  
  a[1] := 0
```

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):  
    let a = [0, 1] in  
    def f(n):  
        a[n] + a[n + 1]  
    in  
    x + f(0)
```

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):  
  def f():  
    [0, 1, 2, 3, 4]  
  in  
  def g(arr, i, j, k):  
    arr[i] * arr[j] * arr[k]  
  in  
  let arr = f() in  
  g(arr, 0, 2, 4)
```

If f allocates in its stack frame and returns a pointer,

The memory will be overwritten by any future calls

Doing this safely would require **copying** any returned data into the caller's stack frame. Not feasible for dynamically sized values.

Stack Allocation

Dynamically sized data can only feasibly be stack allocated if it is **local** to the function, i.e., only used in call stacks that contain the current function's stack frame.

If the dynamically sized data is **returned** from the function that allocates it, we instead allocate it in a separate memory region, the **heap**, and return a pointer to it.

Heap Allocation

The heap contains data whose lifetime is not tied to a local stack frame.

This makes the usage of the data more flexible, but complicates the question of when the data is **deallocated**.

For today, let's assume we do not deallocate memory.

A strategy used in some specialized applications (missiles)

Today's simple heap model: the heap is a large region of memory, disjoint from the stack, some of it is used, and we have a pointer to the next available portion of memory.

Heap Allocation

The heap contains data whose lifetime is not tied to a local stack frame.

This makes the usage of the data more flexible, but complicates the question of when the data is **deallocated**.

For today, let's assume we do not deallocate memory.

A strategy used in some specialized applications (missiles)

The Heap

Let's take a particularly simple view of the heap for now: the heap is a large region of memory, disjoint from the stack. Some amount of this space is used, and we have a **heap pointer** that points to the next available region.

If memory is never deallocated (but also in copying gc), the structure is similar to the stack: we have a region of used space and a region of free space and the **heap pointer**, like the stack pointer, points to the beginning of the free space.

While the stack grows downward in memory, the heap grows upward.

Memory Management

Need our assembly programs to have access to the heap pointer at all times.

We will implement management of the heap in our **runtime system**, i.e., in Rust. Our assembly code programs will interface with the runtime system by calling functions the runtime system provides.

Implementing Arrays

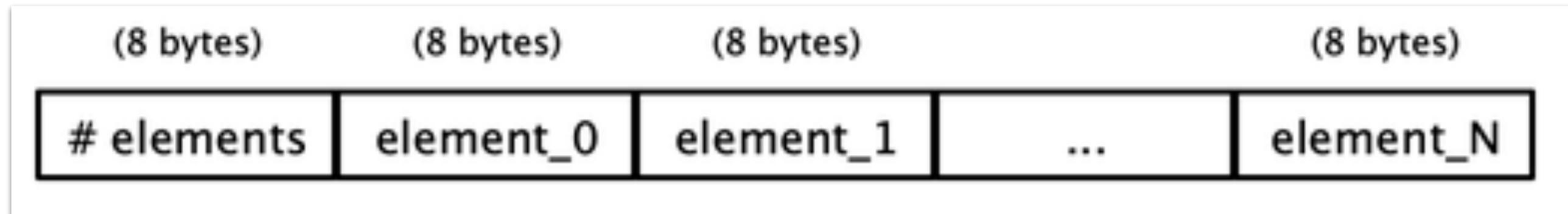
When we implement arrays, we have two different representations to define:

1. How they are stored as "objects" in the heap
2. How they are represented as Snake values

Arrays as Objects

What data does an array need to store?

1. Need to layout the values sequentially so we can implement get/set
2. Need to store the **length** of the array to implement length as well as bounds checking for get/set.



Arrays as Values

The Snake value we store on the stack for an array is a **tagged** pointer to the array stored on the heap.

We overwrite the 2 least significant bits of the pointer with the tag 0b11.

This is safe, as long as those 2 least significant bits of the pointer contain no information, i.e., if they are always 0.

2 least significant bits of a pointer are 0 means the address is a multiple of 4, meaning the address is at a 4-byte alignment.

All arrays on our heap take up size that is a multiple of 8 bytes, so as long as the base of the heap is 4-byte aligned, we maintain this invariant.

Heap/Runtime Demo

Live code: runtime system

Heap/Runtime Demo

Summary:

Pre-allocate a large chunk of memory for our Snake program to use as its heap.

Allocation is managed by the runtime system, i.e., the stub.rs code.